TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

PROGRAMOVÁNÍ POMOCÍ GRAFICKÝCH SYMBOLŮ

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Tomáš Horák**
Vedoucí práce: Ing. Tomáš Martinec, Ph.D.

Liberec 2014

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

PROGRAMMING USING GRAPHICAL SYMBOLS

Diploma thesis

Study programme: N2612 – Electrical Engineering and Informatics
Study branch: 1802T007 – Information Technology

Author: **Bc. Tomáš Horák**
Supervisor: Ing. Tomáš Martinec, Ph.D.

Liberec 2014

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2013/2014**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Tomáš Horák**
Osobní číslo:	**M12000203**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Programování pomocí grafických symbolů**
Zadávající katedra:	**Ústav mechatroniky a technické informatiky**

Zásady pro vypracování:

1. Realizujte nové prvky a funkce do stávajícího vývojového prostředí, které byly navrženy v rámci předcházejícího projektu, a to zejména: lepší podporu různých cílových platforem a podporu strukturovaných datových typů (polí).
2. Dále doplňte do tohoto vývojového prostředí i možnost tvorby aplikací s grafickým uživatelským rozhraním a možnost simulace vytvořených diagramů.
3. Otestujte a zdokumentujte nové vlastnosti vývojového prostředí a vytvořte vhodnou sadu ukázkových příkladů.

Rozsah grafických prací:	**dle potřeby dokumentace**
Rozsah pracovní zprávy:	**40–50 stran**
Forma zpracování diplomové práce:	**tištěná/elektronická**

Seznam odborné literatury:

[1] **ČSN ISO 5807. Zpracování informací DOKUMENTAČNÍ SYMBOLY A KONVENCE PRO VÝVOJOVÉ DIAGRAMY TOKU DAT, PROGRAMU A SYSTÉMU, SÍŤOVÉ DIAGRAMY PROGRAMU A DIAGRAMY ZDROJŮ SYSTÉMU**

[2] **BAYER, Jürgen. C# 2005 - Velká kniha řešení. Brno: Computer Press, 2007, ISBN 978-80-251-1620-3.**

[3] **HORÁK, Tomáš. Programování pomocí grafických symbolů. Liberec, 2012. Bakalářská práce. Technická univerzita v Liberci. Vedoucí práce Ing. Tomáš Martinec, Ph.D.**

Vedoucí diplomové práce:	**Ing. Tomáš Martinec, Ph.D.** Ústav mechatroniky a technické informatiky
Konzultant diplomové práce:	**Ing. Martin Vlasák** Ústav mechatroniky a technické informatiky

Datum zadání diplomové práce:	**10. října 2013**
Termín odevzdání diplomové práce:	**16. května 2014**

prof. Ing. Václav Kopecký, CSc.
děkan

Technická univerzita v Liberci - 14 -
L.S.

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci dne 10. října 2013

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

Podpis:

Poděkování

Na tomto místě bych chtěl poděkovat vedoucímu diplomové práce Ing. Tomášovi Martincovi, Ph.D. za čas, který mi věnoval a odborné rady, které mi velice pomohly při vypracovávání této práce. Dále pak mé rodině, za vytvořené zázemí a podporu při studiu.

Abstrakt

Cílem práce je implementace rozšíření stávající aplikace, která umožňuje tvorbu algoritmů pomocí vývojových diagramů. Tato rozšíření obohatí aplikaci o možnost tvorby složitějších programátorských konstruktů, rozšíří možnosti výstupu a také umožní jednoduchou vizualizaci průběhu vytvořeného algoritmu. Doplněn bude také manuál, který je součástí aplikace. V neposlední řadě budou vytvořeny ukázkové úlohy využívající nové funkce. V teoretické části práce budou popsány způsoby programování nad systémem Windows Presentation Foundation.

Klíčová slova

grafický symbol, programování, vizuální programovací jazyk, vývojové prostředí, vývojový diagram

Abstract

The goal of this thesis is to implement extensions of an existing application which allows creating algorithms with flowcharts. These extensions enrich the application with a possibility of creating more complex programming constructs, a better output and allow a simple visualization of a process of created algorithm. A manual, which is a part of the application, will be amended as well. Finally there will be created sample tasks, which use new functions. In the theoretical part there will be described ways of programming on the Windows Presentation Foundation.

Keywords

graphical symbol, programming, visual programming language, development environment, flowchart

Obsah

Seznam obrázků

Seznam zkratek

API – application programming interface

CLR – common language runtime

FBD – function block diagram

GUI – graphical user interface

LAD – ladder diagram

PLC – programmable logical controller

SDCC – small device C compiler

UML – unified modeling language

WPF – windows presentation foundation

XAML – extensive application markup language

Úvod

Vývojové diagramy zná asi každý člověk, který se učil programovat v nějakém kurzu či ve škole. Jejich předností je přehledné vizuální zobrazení typu operace nad daty a všech dostupných cest algoritmu, čehož se hojně využívá právě při výuce. Ten, kdo někdy navrhoval algoritmus pomocí vývojového diagramu, ale také určitě narazil na několik nevýhod. Pokud je diagram tvořen na papíře, je velice těžké již nakreslené části změnit a pokud ke změnám přeci jen dojde, diagram velmi rychle ztrácí svou přehlednost. Tato nevýhoda ale mizí, pokud uživatel k návrhu použije počítač a nějaký vhodný software.

Původní zadání, na které tato práce navazuje, byl návrh grafického jazyka, který by umožnil navrhovat jednoduché algoritmy tak, aby tento úkol byl zvládnutelný i pro mladší žáky, případně amatéry v oblasti programování. Na základě rešerší byly vybrány vývojové diagramy, některé důvody tohoto výběru plynou z předchozího odstavce.

Po výběru jazyka byly navrhnuty základní funkce vývojového prostředí, které umožní propojit svět vývojových diagramů se serióznějším programováním pomocí zažitých textových programovacích jazyků. Již samotná možnost existence tohoto propojení značí, že i přes to, že vývojové diagramy jsou mnoha lidmi považovány za přežitek, jsou velmi mocným nástrojem pro programování. Mezi základními funkcemi prostředí byla možnost navrhnutí jednoduchého algoritmu pomocí vývojového diagramu, definice proměnných a přiřazení funkcí použitým symbolům diagramu. Prostředí následně umožňovalo diagram zpracovat a vygenerovat zdrojový kód v konvenčních programovacích jazycích. Aby byla názornost ještě umocněna, byla přidána možnost zdrojový kód zkompilovat a vytvořit fungující aplikaci.

Aplikace, která byla z návrhů vytvořena, měla ovšem obrovský potenciál k rozšíření. Těmito rozšířeními se zabývá tato práce. Původní forma, kdy v jednom projektu bylo možné navrhnout pouze jeden algoritmus, který představoval jednu jedinou funkci, byla velice omezující. Bude popsán způsob, jak přidat do aplikace možnost tvorby libovolného množství funkcí, které mohou být vzájemně provázány. Dalším nedostatkem, který bude vyřešen, je nutnost použití pouze jednoduchých datových typů proměnných. Důvodem je, že hodně učebnicových algoritmů pracuje například nad poli, příkladem můžou být algoritmy řadící, či vyhledávací. Velkým rozšířením, které bude navrhnuto a implementováno, je simulátor. Simulátor má velký potenciál pro nastínění práce vytvořených algoritmů, protože uživatel má možnost nahlédnout do jejich průběhu a vnitřních stavů při práci nad reálnými daty. Původní aplikace také umožňovala kompilaci vytvořených algoritmů jako konzolových programů. Konzolové programy už ovšem nejsou v dnešní době pro laické uživatele vůbec přitažlivé, proto bude implementována možnost vytvořit aplikaci s grafickým uživatelským rozhraním, kde ovšem veškerá logika bude definována pomocí vývojových diagramů. To výsledku přidá pro tyto uživatele ještě větší atraktivitu.

Tato práce je rozdělena na čtyři části, v první části jsou krátce popsány výsledky předchozích prací, následuje představení technologie použité v této práci, popis rozhraní vytvořené aplikace, jehož rozšířená verze bude součástí přiloženého manuálu a poslední část obsahuje některé zajímavé implementační detaily, které se vyskytly při psaní aplikace.

1 Důvody vzniku práce a úvod do problematiky

1.1 Bakalářský projekt

Prvotní zadání, týkající se tématu programování pomocí grafických symbolů, vzniklo jako zadání bakalářského projektu. Cílem práce bylo nalezení oborů, ve kterých jsou použity grafické jazyky pro vyjádření algoritmů, dále rešerše norem pro využívání grafických symbolů k vyjádření algoritmu, rešerše dostupných programovacích jazyků, které využívají grafických symbolů jako prostředku pro programování a návrh systému pravidel a symbolů, který by umožnil programování jednoduchých aplikací i pro žáky základních škol.

V projektu byly popsány 4 grafické jazyky:

- Diagram funkčních bloků (dále jen FBD), ošetřený normou IEC 61131-3
- Žebříkový diagram (dále jen LAD), popsaný touže normou
- Stavový diagram, ošetřený specifikací jazyka UML
- Vývojový diagram s normou ČSN ISO 5807, která byla rozepsána podrobněji

Bylo popsáno také 8 vývojových prostředí. Pro jazyk FBD to byly programy Siemens Logo! a Picaxe, pro jazyk LAD také program Siemens Logo!, pro stavový diagram program Qfsm a pro vývojové diagramy programy Flowol, Picaxe a RoboPro. Dále byly popsány dvě prostředí s vlastním jedinečným způsobem zobrazení algoritmu a to programy Baltík a Scratch.

Poslední částí projektu byl návrh vlastního vývojového prostředí pro grafické programování. Jako jazyk byly zvoleny vývojové diagramy dle normy ČSN ISO 5807 s navrženými drobnými odchýleními. Byl navržen systém tvorby vývojových diagramů spočívající v přidávání symbolů na pracovní plochu metodou *drag and drop*, jejich spojování a vyplňování atributů pro konkrétní symboly. Návrh také posuzoval možnost zpracovat vytvořené vývojové diagramy a vygenerovat zdrojový kód v nějakém klasickém programovacím jazyce. Okrajově byly popsány možnosti vytvoření simulátoru algoritmů a tvorba funkcí.

1.2 Bakalářská práce

Na projekt navazovala bakalářská práce, jejímž hlavním cílem bylo shrnutí předchozích rešerší, konečný popis zvoleného grafického jazyka a hlavně vytvoření funkční aplikace z předchozího návrhu.

V práci byla snaha zaměřit se na výhody, nevýhody a možné problémy při použití grafických jazyků při návrhu algoritmů s důrazem na vývojové diagramy.

Byly podrobně popsány vývojové diagramy dle normy ČSN ISO 5807 a rozebrány jednotlivé symboly, použité v budoucí aplikaci. Byly to:

- Mezní značka – symbol pro začátek a konec algoritmu
- Spojnice – určuje, jak jsou symboly na sebe napojeny
- Data – představuje vstup nebo výstup dat z algoritmu
- Zpracování – symbolizuje nějakou operaci nad daty
- Příprava – určuje začátek a typ cyklu
- Rozhodování – představuje podmínku

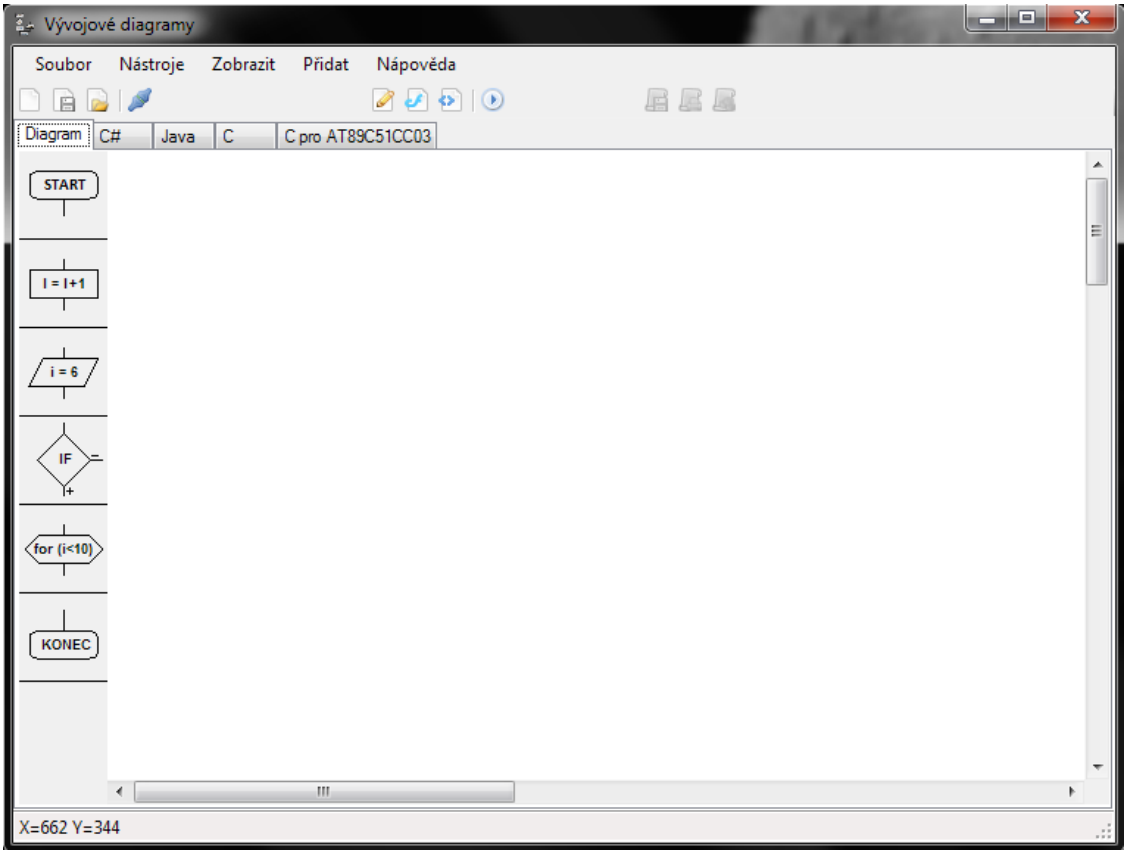
Mimo popsání samotných symbolů bylo nutné definovat některé další prvky aplikace. Každému symbolu byl jasně definován tvar textového popisu, který je jeho nedílnou součástí, pro symbol podmínky byla definována dostupná porovnávací pravidla a byl definován způsob práce s proměnnými a jejich datové typy. Dostupné datové typy byly:

- Celá čísla (*integer*)
- Desetinná čísla (*double*)
- Logické proměnné (*boolean*)
- Znaky (*char*)
- Řetězce znaků (*string*)

Zbytek práce popisoval postup implementace výsledné aplikace. Byla vytvořena aplikace umožňující:

- Tvorbu vývojových diagramů na pracovní ploše.
- Specifikování konkrétní operace pro každý symbol.
- Vytvoření proměnných nutných k práci.
- Jednoduchou kontrolu kompletnosti a logické návaznosti diagramu.
- Z vytvořeného diagramu generování zdrojového kódu v jazyce C#, Java a C.
- Generování spustitelných programů za pomoci externího překladače jazyka C#.
- Za pomoci speciálních proměnných generování zdrojových kódů programů pro vývojovou desku s procesorem Atmel AT89C51CC03 v jazyce C.
- Ze zdrojového kódu generování binárního souboru vhodného k nahrání do této desky.
- Ukládání a opětovné načítání vytvořeného projektu.
- Ukládání vygenerovaných kódů v textovém formátu.
- Tisk diagramů i kódů.

Aplikace měla i velké množství nedostatků, resp. zjednodušení. Úplně chyběla možnost tvorby polí, nebylo umožněno tvořit funkce, což ale nebylo tak omezující vzhledem k tomu, že cílem bylo umožnění tvorby pouze jednoduchých algoritmů. Nebyl vytvořen dříve navržený simulátor algoritmů na úrovni jednotlivých symbolů. A také byla pro vytvoření programu použita zastaralá grafická knihovna Windows Forms.



Obrázek 1: Hlavní okno aplikace

Prostředí bylo jednoduché s dominantní komponentou se záložkami uprostřed. První záložka obsahovala pracovní plochu pro tvorbu diagramů a vlevo dostupné symboly budoucího diagramu. Byly to symboly:

- Start
- Zpracování
- Vstup a výstup dat
- Podmínka
- Cyklus
- Konec

Na dalších záložkách byly zobrazeny zdrojové kódy vygenerované z vytvořeného diagramu.

Aplikace umožňovala kompilaci vygenerovaných zdrojových kódů pro obě platformy. Pro konzolové aplikace to byl kompilátor jazyka C#, který je součástí každé instalace .NET Framework. Zde konkrétně byl použit kompilátor verze 2.0, protože tato verze je obsažena i v každé instalaci Windows XP. Kompilátor byl volán ručně ze svého umístění, což mělo velkou nevýhodu, uživatel musel ručně nastavit cestu k jeho souboru. Kompilace programů pro vývojovou desku probíhala

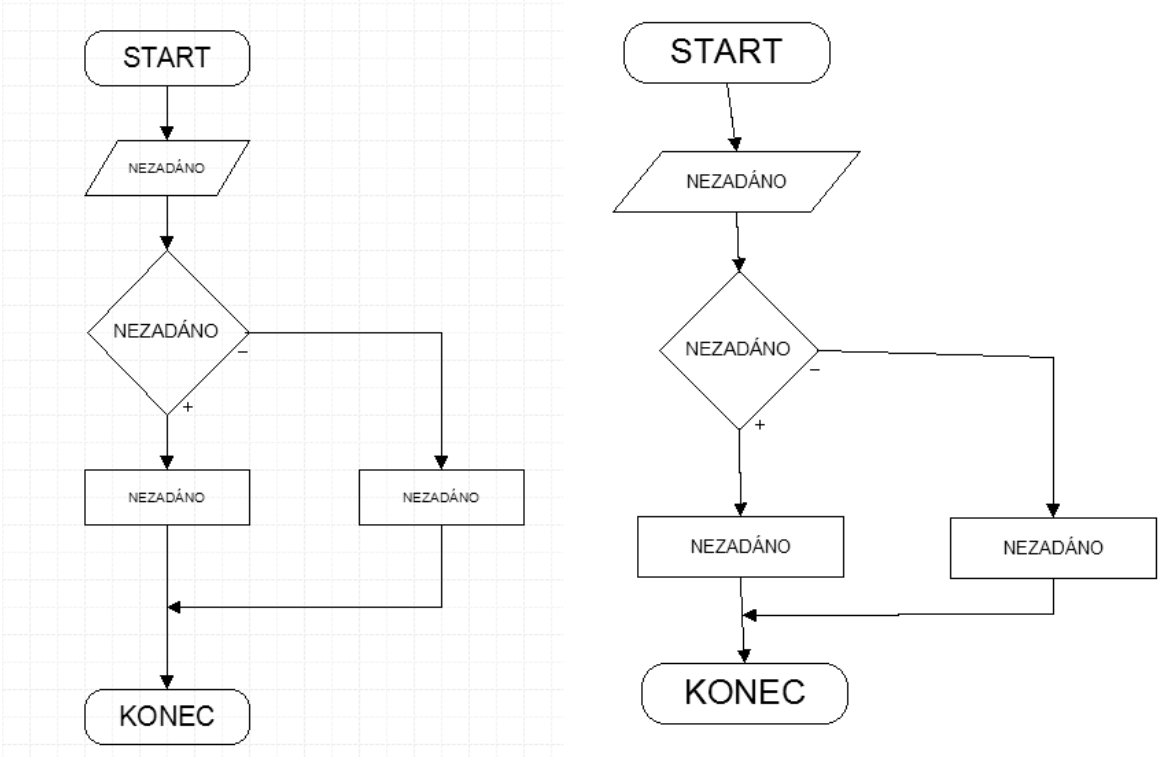
obdobně. Zde byl použit open source kompilátor Small Device C Compiler (dále jen SDCC) jazyka C. Uživatel musel tento kompilátor nainstalovat a také k němu zadat cestu v aplikaci.

Součástí práce byla dokumentace k vzniklému prostředí s podrobným popisem všech funkcí a možností a sada ukázkových úloh, jak konzolových aplikací, tak aplikací pro vývojovou desku [1].

1.3 Diplomový projekt

Cílem diplomového projektu byl návrh rozšíření původního programu a vytvoření podkladu pro tuto práci. Některá rozšíření byla zčásti i implementována.

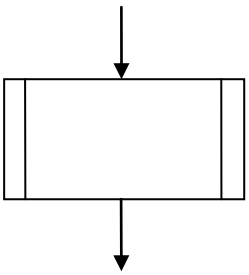
První změnou bylo přepracování umísťování jednotlivých grafických symbolů na pracovní ploše. V původním programu bylo umísťování navrženo zcela volně, s čímž souvisel velký problém, bylo velice těžké vytvořit diagram tak, aby byly všechny čáry a symboly rovnoběžné s osami. Z tohoto důvodu byla na pracovní plochy přidána aktivní mřížka, ke které se všechny symboly a spojnice přichycují. Mřížky jsou ve své podstatě dvě, jedna hrubší, ke které jsou zarovnány symboly a jedna jemnější, ke které se zarovnávají spojnice.



Obrázek 2: Příklad diagramu, vlevo s mřížkou, vpravo bez mřížky

Další navrženou změnou bylo umožnění tvorby funkcí použitelných v algoritmech. Důvodem byla snaha umožnit tvorbu komplexnějších algoritmů, umožnění použití funkcí také zpřístupní složitější programátorské konstrukce, například rekurzi.

Byl navržen nový symbol, který symbolizuje použití funkce. Symbol má tvar obdélníku se zdvojenými stranami, norma [2] tento symbol popisuje jako symbol pro podprogram.



Obrázek 3: Symbol pro podprogram

Mimo symbolu bylo nutné také navrhnout úpravu grafického rozhraní programu. Bylo navrženo rozšíření stávající grafické komponenty pro tvorbu algoritmu spočívající v tom, že samotná komponenta byla umístěna do panelu se záložkami tak, že každá záložka bude symbolizovat jednu funkci. Dále byl navržen formulář pro správu vlastností jednotlivých funkcí, lokálních proměnných a parametrů.

Stávající program uměl vytvářet algoritmy pro dvě platformy, konzolové aplikace a aplikace pro vývojovou desku s mikroprocesorem Atmel. Bylo navrženo přidání třetí platformy a to aplikací s grafickým uživatelským rozhraním. Důvodem zvolení aplikací s grafickým rozhraním a ne například simulátoru programovatelných automatů (PLC) byla myšlenka, že umožnění uživatelům, kterými měli být hlavně začátečníci nebo děti, vytvoření fungující aplikace spustitelné na jakémkoliv počítači s Windows bude více atraktivní a podnítí jejich kreativitu.

S tvorbou grafických aplikací se váže nutnost vytvoření grafického rozhraní, ke kterému by se přidávala funkčnost. Byl velmi jednoduše navržen vzhled editoru grafického rozhraní aplikace a také seznam dostupných komponent, ze kterých by se grafické rozhraní tvořilo. Byly to:

- Okno (*Window*)
- Tlačítko (*Button*)
- Popisek (*Label*)
- Výběrové tlačítko (*RadioButton*)
- Zobrazení průběhu (*ProgressBar*)
- Vstup textu (*TextBox*)
- Časovač (*Timer*)

Dle návrhu budou výstupem grafické aplikace nad grafickou knihovnou Windows Presentation Foundation, to znamená, že samotná grafika bude překládána do jazyka XAML a její logika bude v jazyce C#.

Bylo navrženo také rozšíření aplikace o možnost tvorby polí, jelikož pole jsou nedílnou součástí složitějších algoritmů. Popsány jsou dvě možnosti implementace a to rozšíření stávající třídy pro proměnnou o popis pole, nebo svázaní více jednotlivých proměnných pomocí názvu a vytvoření pole.

Posledním navrženým rozšířením byl simulátor naprogramovaných algoritmů na úrovni jednotlivých grafických symbolů, případně řádků vygenerovaného kódu. Možnost simulování průchodu algoritmem velice názorně ukáže procesy, kterými data prochází a vnitřní stavy algoritmu.

Byly popsány jednotlivé důležité funkce pro simulaci, jako je krokování, automatický posun v časovém intervalu, zobrazení aktuálních hodnot proměnných, zobrazení aktuální pozice v diagramu a kódu atd. [3].

2 Technologie nově vzniklé aplikace

Navržená rozšíření z předchozí práce jsou tak rozsáhlá, že nebylo možné předchozí aplikaci pouze rozšířit a bylo nutné jí celou přepsat. S tím se váže i použití nové technologie. Původní aplikace byla napsána nad grafickou knihovnou Windows Forms, nově vzniklá aplikace využívá knihovny Windows Presentation Foundation (dále jen WPF).

2.1 Úvod

Technologie WPF byla představena s .NET Framework 3.0, který je součástí systémů Windows Vista a vyšších. Po dodatečné instalaci je dostupný i ve Windows XP se Service Pack 2. Grafické jádro WPF je založeno na vektorové grafice, je nezávislé na rozlišení monitoru a je akcelerované grafickou kartou [4].

2.2 Struktura aplikace

Každá WPF aplikace potřebuje pro svou činnost 2 hlavní objekty. Objekt okna typu *Window* s nejdůležitější metodou *Show()*, která okno zobrazí, a objekt aplikace typu *Application* s metodou *Run()*, která vytvoří tzv. cyklus zpráv, ve kterém jsou přijímány vstupy od uživatele [5].

Objekt *Application* je unikátní v každé aplikaci a nemůže existovat více jeho instancí najednou. Může být vytvořen dříve než okno aplikace, ale jeho metoda *Run()* musí být zavolána až poté, co okno existuje a buď je již zobrazeno, nebo je předáno metodě *Run()* jako parametr. Objekt *Application* zapouzdřuje důležitá data aplikace jako je její životní cyklus, přístup k hlavnímu oknu, přístup ke zdrojům, parametry příkazového řádku, ukončovací kódy, navigace u stránkových aplikací atd. [6]. Mimo dat jsou v něm definovány některé důležité události v životním cyklu aplikace, například *OnStartup*, vyvolaná po volání metody *Run(), OnExit* při ukončování aplikace, *OnSessionEnding* když uživatel ukončuje přihlášení k relaci systému.

2.3 Dispatcher

Dispatcher je objekt jedinečný pro každou WPF aplikaci. Dalo by se říci, že je to fronta událostí aplikace, jejím úkolem je zajištění zpracování událostí v rámci hlavního vlákna. Tyto události mohou být například akce od uživatele, nebo nutnost překreslení grafiky. *Dispatcher* tedy zajišťuje, že všechnu práci s grafikou okna vykonává jedno vlákno a nemůže dojít k tomu, že by dvě vlákna pracovala s jednou komponentou zároveň.

Velkou nevýhodou tohoto přístupu je zastavení celé aplikace, pokud hlavní vlákno vykonává složitou, nebo časově náročnou úlohu. Ve chvíli, kdy probíhá její výpočet, aplikace přestane reagovat na všechny vstupy od uživatele až do té doby, dokud složitý výpočet neskončí. Pokud se takto složité, nebo časově náročné úlohy v aplikaci vyskytují, je nutné je řešit v jiném vlákně. Složitý výpočet se přesune do jiného vlákna a hlavní vlákno se může dále věnovat obsluze aplikace. Problém nastává,

pokud chceme zobrazit výsledek výpočtu z vedlejšího vlákna. Vedlejší vlákno nemůže přímo přistupovat ke grafickým komponentám, protože ty řídí vlákno hlavní, resp. *Dispatcher*. Vedlejší vlákno tedy musí požadavek na zobrazení výpočtu (překreslení grafiky) zařadit do fronty *Dispatcheru* a ten jej zpracuje sám.

Všechny grafické komponenty ve WPF mají jako svého předka třídu *DispatcherObject*. Třída *DispatcherObject* zajišťuje, že každá komponenta si uchovává referenci na objekt *Dispatcher*, který se stará o její funkčnost. Dále obsahuje metodu pro zjištění, zda ke komponentě přistupujeme z vlákna, o které se stará požadovaný *Dispatcher*, tzn. z hlavního vlákna a metodu, která vyvolá výjimku, pokud tomu tak není. Tato metoda je volána automaticky při většině operací s komponentou [7].

2.4 Architektura

O vykreslování veškeré grafiky WPF se na nejnižší úrovni stará *DirectX*. Jako mezivrstva mezi ním a knihovnami WPF funguje *Media Integration Layer* (*milcore*), která běží mimo .NET virtuální stroj. S *milcore* komunikuje .NET knihovna *PresentationCore*, která už běží pod .NET virtuálním strojem. Nad *PresentationCore* leží *PresentationFramework*, který ji ještě rozšiřuje [8].

2.4.1 Objektový model grafické komponenty

Základní třídou pro každou grafickou komponentu je již dříve zmíněný *DispatcherObject*, starající se o komunikaci s *Dispatcherem*. Nad *DispatcherObject* leží třída *DependencyObject* rozšiřující objekt o základní prvky *Data Bindingu*. Následuje třída *Visual*, která se stará o základní vykreslování, testování rozsahu komponenty a komunikaci s *milcore*. Nad třídou *Visual* je třída *UIElement*, která přidává možnosti pozicování a vstupu od uživatele. Následuje třída *FrameworkElement*, přidávající podporu stylování a šablonování. Řetězec uzavírá třída *Contol*, která umožňuje předpřipravené šablonování vzhledu odvozených komponent.

2.5 Jazyk XAML

Technologie WPF důsledně odděluje návrh grafického prostředí aplikace (dále jen GUI) od samotné logiky aplikace. Jazyk XAML je značkovací jazyk založený na XML pro tvorbu GUI. Všechny grafické prvky aplikace jsou popsány tímto jazykem, i když jsou navrženy nějakým editorem grafického rozhraní, například editorem integrovaným v Microsoft Visual Studiu. Syntaxe je podobná XML a je velmi intuitivní, příkladem může být tlačítko na panelu [5][9][10].

```
<StackPanel>
  <Button Content="Klikni"/>
</StackPanel>
```

```
<StackPanel>
  <Button>Klikni</Button>
</StackPanel>
```

Obrázek 4: Definice tlačítka

Vlastnosti grafického prvku jsou připisovány jako atributy elementu, nebo pomocí vnořeného elementu, jehož název má tvar <*jméno_komponenty.vlastnost*>

```
<Button Background="Blue" Foreground="Red" Content="Klikni"/>
      <Button.Background>
            <SolidColorBrush Color="Blue"/>
      </Button.Background>
      <Button.Foreground>
            <SolidColorBrush Color="Red"/>
      </Button.Foreground>
      <Button.Content>
            Klikni
      </Button.Content>
</Button>
```

Obrázek 5: Definice vlastností tlačítka

Příklad ukazuje zápis vlastností tlačítka jak pomocí zápisu přes atribut elementu, tak pomocí elementu vnořeného.

Interaktivita je grafickým prvkům přiřazována pomocí událostí. Každý grafický prvek má definovanou množinu událostí, na které reaguje. Příkladem může být například tlačítko s událostí vyvolanou po kliku myší. Logika, kterou má program vyvolat po události, je komponentě přiřazena pomocí atributů, jejichž jméno je stejné jako jméno události a parametrem je název metody, která se má zavolat.

```
<Button Content="Klikni" Click="onClick"/>
```

Obrázek 6: Definice reakce na událost

2.5.1 Obory názvů XAML

Obor názvů definuje, co jednotlivé použité elementy jazyka XAML představují. Je deklarován atributem *xmlns* elementu a platí pro samotný element a elementy do něj vnořené. Pokud je deklarován samotným atributem x*mlns*, jedná se o implicitní deklaraci a není nutné uvádět její jméno, implicitní obor názvů může být ale pouze jeden. Pokud je nutné použít oborů více, musí se za *xmlns* uvést jméno. Výsledek má potom tvar *xmlns:jmeno*. Jako parametr atributu *xmlns* se uvádí jednoznačný název oboru. Jako název oboru často slouží URL adresa. Pro elementy jazyka XAML slouží obor názvů s adresou *http://schemas.microsoft.com/winfx/2006/xaml/presentation*. Aby tedy bylo možné

použít element <*Button*>, musí být tlačítko uvnitř nějakého elementu s definovaným implicitním oborem názvů jazyka XAML.

```
<Window  xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation>
     <Button Content="Klikni" Click="onClick"/>
</Window>
```

Obrázek 7: Definice oboru názvů

2.6 Nezávislost na rozlišení

Jak již bylo zmíněno, zobrazování ve WPF není závislé na fyzickém rozlišení zobrazovacího média, to znamená, že ať je prvek zobrazený na jakkoli jemném displeji, jeho fyzická velikost vždy zůstane stejná. Toho je docíleno tím, že rozměry nejsou zadávány v pixelech, ale v *device independent pixels* (dále jen pt). Jeden takový „pixel“ má rozměr 1/96 palce (cca 0.26 mm). V praxi to znamená, že okno o rozměrech 288×192 pt má při rozlišení 96 dpi (dots per inch) velikost 3×2 palce. Pokud se rozlišení změní na 120 dpi, změní se jeho rozměry ve fyzických pixelech, ale velikost v palcích zůstane stejná [5].

Problém může nastat při použití bitmapových obrázků. Se škálováním rozhraní se škálují i obrázky a může dojít k rozmazávání.

2.7 Vykreslování

Grafika WPF je vektorová, akcelerovaná grafickou kartou a o vykreslování se stará *DirectX*. V porovnání se starší knihovnou Windows Forms, kde se o vykreslování starala knihovna *GDI+*, která grafiku počítá na procesoru. Použitím WPF lze tak dosáhnout několikanásobného zrychlení. I způsob vykreslování grafických primitiv se změnil. Pokud chtěl programátor používající Windows Forms vykreslit například obdélník, příkaz k jeho vykreslení musel zapsat do metody *OnPaint* komponenty, do které chtěl kreslit. Ve WPF už tento postup neplatí. I taková primitiva jako je obdélník jsou rovnocenná s ostatními grafickými prvky a jejich zápis v XAML je obdobný. Dají se tudíž u nich použít stejně pokročilé funkce, jako u klasických grafických komponent. Příkladem může být testování kliku či pohybu myši, pozicování, nebo vstup z klávesnice [5].

Vektorový přístup při vykreslování má i další výhody. Přístupné jsou grafické transformace, jako je například zmenšení či rotace, po jejichž aplikaci se nesníží kvalita vykreslování.

Všechna dostupná grafická primitiva jsou odvozena z třídy *Shape*, dostupná jsou:

- *Ellipse* (elipsa)
- *Line* (jednoduchá úsečka)
- *Path* (křivka)
- *Polygon* (n-úhelník)

- *Rectangle* (obdélník)
- *Polyline* (více úseček kreslených najednou)

Mimo geometrie jsou nejdůležitějšími vlastnostmi grafických primitiv štětce v atributech *Stroke* a *Fill*. *Stroke* určuje typ čáry, případně hrany a *Fill* určuje typ výplně.

2.8 Data Binding

Data Binding, neboli česky datová vazba, je velmi silný nástroj pro spojení grafických komponent s jejich myšleným obsahem, který může být uložen v jednoduché proměnné, nebo třeba i rozsáhlé databázi. Velká síla datových vazeb je v tom, že slouží jak pro zobrazení dat, tak i pro jejich editaci. To znamená, že pokud je spojena textová proměnná s grafickou komponentou, umožňující editovat text, je obsah této proměnné zobrazen v komponentě a pokud uživatel text v komponentě přepíše, obsah je automaticky změněn i v proměnné. To velmi často může nahradit tvorbu události a tím zpřehlednit kód [5].

Dva subjekty spojené datovou vazbou se nazývají zdroj a cíl. Zdrojem bývají nějaká data a cílem grafická komponenta. Toto ale není pravidlem, zdrojem dat může být například i navolená hodnota na posuvníku (*ScrollBar*) a cílem třeba ukazatel průběhu (*ProgressBar*).

```
<StackPanel>
    <ScrollBar Name="sBar" Maximum="100" />
    <ProgressBar Value="{Binding ElementName=sBar, Path=Value}" />
</StackPanel>
```

Obrázek 8: Data binding

Kód v příkladu ukazuje navázání hodnoty posuvníku na hodnotu ukazatele průběhu, nepotřebuje žádnou další logiku a po kompilaci se s posuvem posuvníku hodnota automaticky zobrazuje na ukazateli průběhu.

Datová vazba ale nemůže pracovat s objekty jakéhokoliv typu. Zdroj dat musí obsahovat statický objekt typu *DependencyProperty*, který zajišťuje ohlášení změny všem cílům. A cíl dat musí být odvozen od třídy *DependencyObject*, což zajistí aktualizaci při změně. Pokud navazujeme grafické prvky, tyto podmínky jsou splněny, pokud ale bude navazován například dříve zmíněný text na komponentu umožňující jeho editaci, musí být *DependencyProperty* vytvořena ručně.

```
//vytvoření DependencyProperty v C#
public string Text
{
     get { return (string)GetValue(TextProperty); }
     set { SetValue(TextProperty, value); }
}
public static readonly DependencyProperty TextProperty =
                 DependencyProperty.Register("Text",
                 typeof(string), typeof(Window),
                 new PropertyMetadata(""));

//navázání na grafickou komponentu v XAML
<TextBox Text="{Binding Path=Text, ElementName=window}" />
```

Obrázek 9: DependencyProperty

Příklad ukazuje vytvoření textové proměnné, k ní vytvoření *DependencyProperty* a následné navázání na text grafické komponenty.

Existují čtyři druhy datových vazeb dle směru aktualizace:

- *OneTime* – aktualizace proběhne pouze jednou
- *OneWay* – aktualizace probíhá pouze při změně zdroje, aktualizuje se obsah cíle
- *OneWayToSource* – aktualizace probíhá pouze při změně cíle, aktualizuje se obsah zdroje
- *TwoWay* – aktualizace probíhá oběma směry

Datové vazby nemusí být realizovány pouze tímto jednoduchým způsobem, v případě nutnosti je možné mezi svázané objekty připojit konvertor, který může měnit průchozí data. Například jejich datový typ, nebo jejich konkrétní hodnoty. Mimo konvertoru je možné vložit i validátor, který může příchozí data kontrolovat a v případě nutnosti aktualizaci dat zrušit a vrátit původní hodnoty.

2.9 Prostředky

Každému elementu jazyka XAML se dá přiřadit slovník konstant, kterému se říká prostředky. Tento postup může být vhodný, například pokud je nutné použít dvě velikosti písma, ale ještě není známo, jaké to budou. Obalovému elementu, ve kterém se tyto dvě velikosti písma budou používat, se definují dva prostředky (konstanty) a dále místo přímých hodnot budou použita jen jejich jména. Výhodou potom je, že pokud se mění hodnota, stačí ji změnit v prostředku a nemusí být přepsána na všech jejích umístěních [5].

```
<StackPanel>
     <StackPanel.Resources>
            <s:Double x:Key="fontVelky">15</s:Double>
            <s:Double x:Key="fontMaly">10</s:Double>
     </StackPanel.Resources>
     <Label FontSize="{StaticResource fontVelky}" Content="Velký text" />
     <Label FontSize="{StaticResource fontMaly}" Content="Malý text" />
</StackPanel>
```

Obrázek 10: Prostředky

Příklad ukazuje nastavení velikosti písem popisků pomocí prostředků.

2.10 Styly

Styly jsou kolekce vlastností, které jsou přiřazovány objektům. Styl může být definován několika způsoby, prvním z nich je definice stylu elementu pomocí vnořeného elementu s názvem tvaru *<název_elementu.style>*. Takový styl bude aplikován pouze na jeden konkrétní element, pro který je definován. Druhým způsobem je definice stylu jako prostředku elementu. S tímto přístupem přicházejí dvě možnosti použití, buď je definován identifikátor stylu pomocí atributu *x:Key* a styl se použije tak, že v elementech, kde se má použít, se použije atribut tvaru *Style="{StaticResource klíč_stylu}"*, nebo se stylu definuje atribut *TargetType="{x:Type Element}"*, čímž docílíme automatické aplikace stylu na všechny vnořené elementy zvoleného typu [5].

Styl může být složen ze dvou typů objektů, jimiž jsou *Setter* a *Trigger*. *Setter* slouží k jednoduchému nastavení požadované vlastnosti a *Trigger* k nastavení podle podmínky. *Trigger* je definován vlastností, na kterou má reagovat, a její hodnotou, s kterou se *Trigger* aktivuje. *Trigger* obsahuje vnořenou množinu objektů typu *Setter*, které se při splněné podmínce aktivují.

```
<Label Content="Text labelu">
   <Label.Style>
      <Style>
         <Setter Property="Label.FontSize" Value="20" />
         <Style.Triggers>
            <Trigger Property="Label.IsMouseOver" Value="True">
               <Setter Property="Label.FontSize" Value="10">
         </Style.Triggers>
      </Style>
   </Label.Style>
</Label>
```

Obrázek 11: Použití stylů

Příklad ukazuje definování stylu popisku. Styl obsahuje jednoduchý *Setter*, který nastaví popisku velikost písma 20 pt. Dále obsahuje *Trigger,* který je aktivován, pokud se nad popiskem nachází kurzor myši a nastaví písmo na 10 pt.

2.11 Šablony

Příkladem použití šablon ve WPF je definice vzhledu grafických komponent. Rozdíl mezi stylem a šablonou je ten, že každý grafický prvek musí mít nějakou šablonu, ale nemusí mít definovaný styl. Výhodou je, že šablona každého prvku se dá nahradit, takže je možné předefinovat vzhled všech grafických prvků používající nahrazenou šablonu. Nevýhodou oproti stylům může být to, že při definování nové šablony musíme definovat celý vzhled komponenty od základů znovu [5].

2.12 Pozicování

Ve WPF se většinou již nepoužívá klasické pozicování pomocí určení pevné pozice elementu, i když je to také možné. Většina rozměrů a pozic elementů se počítá dynamicky dle velikosti okna, nebo obsahu.

Většina grafických prvků má atributy *Width* a *Height,* tedy šířku a výšku. Při prvním přístupu by se jim pevně přiřadila hodnota například v *pt*. Druhou možností je přiřadit rozměrům hodnotu *Auto* a nechat aplikaci, aby rozměry dopočítala. Při tomto přístupu je ale nutné určit chování objektu vůči nadřazeným objektům a obsahu. K tomuto slouží atributy *VerticalAlignment* a *HorizontalAlignment,* jejichž přípustné hodnoty jsou [11]:

- *Left* (jen u horizontálního zarovnání) – prvek je zarovnán k levému okraji rodičovského elementu
- *Right* (jen u horizontálního zarovnání) – prvek je zarovnán k pravému okraji rodičovského elementu
- *Top* (jen u vertikálního zarovnání) – prvek je zarovnán k hornímu okraji rodičovského elementu
- *Bottom* (jen u vertikálního zarovnání) – prvek je zarovnán ke spodnímu okraji rodičovského elementu
- *Center* – prvek je zarovnán na střed
- *Stretch* – prvek bude roztažen na celou šířku/výšku rodičovského elementu

S pozicováním souvisí také dva důležité atributy *Margin* a *Padding*. *Margin* určuje, kolik místa zůstane okolo zvoleného elementu, tedy vnější okraj a *Padding* určuje odstup grafiky elementu od jeho hranic, tedy vnitřní okraj. Oba tyto atributy jsou typu *Thickness*, to znamená, že určují hodnoty pro všechny čtyři okraje. Typ *Thickness* v sobě zapouzdřuje 4 číselné hodnoty.

2.13 Layouty

Layouty silně souvisí s pozicováním, jsou to vlastně panely, které svým chováním určují rozvržení komponent v okně a reakce na změnu rozlišení, nebo rozměrů okna. Ve WPF existuje více typů layoutů [12]:

- *Canvas* – umožňuje pozicování pomocí absolutních souřadnic
- StackPanel – komponenty v tomto kontejneru jsou pozicovány horizontálně nebo vertikálně jeden za druhým, ale vždy v jedné řadě/sloupci
- *WrapPanel* – obdobný jako *StackPanel* ale umožňuje více řad/sloupců
- *DockPanel* – umisťování se provádí připínáním komponent k okrajům kontejneru, možnosti jsou:
  - *Bottom*
  - *Left*
  - *Right*
  - *Top*
- *Grid* – vytvoří mřížku. U jednotlivých komponent je nutné určit pozici buňky, do které patří

Layouty je možné do sebe dále vnořovat a vytvářet tak složitější struktury. Další možností ošetření změny rozměrů je povolení scrollování obsahu kontejnerů, jejichž rozměry není vhodné nadále zvětšovat. Toho se dá docílit jednoduše tak, že do layoutu se vloží *ScrollViewer* s definovanými maximálními rozměry, při jejichž překročení se zobrazí S*crollBar*.

2.14 Příkazy (Commands)

Command ve WPF je jasně definovaná akce, která ale může být vyvolána více podněty od uživatele. Příkladem může být příkaz pro vložení textu do schránky, ten může být vyvolán všeobecně známou kombinací kláves *Control+C*, ale také například kontextovým menu vyvolaným myší, nebo ikonou na hlavním panelu aplikace. Z programátorského hlediska stačí napsat obslužnou metodu *Execute*, která se má vykonat, pokud je *Command* aktivován a dále nastavit příslušným grafickým prvkům atribut *Command* [13].

Další užitečnou vlastností je možnost deaktivovat *Command*, respektive *Command* si sám testuje, zda se může vykonat pomocí metody *CanExecute()*. Pokud není *Command* aktivní, zároveň s ním jsou automaticky deaktivovány příslušné grafické komponenty, které ho můžou vyvolat. Pokud zmíněnou komponentou je tlačítko, nebo položka menu, uživatel má přehled o tom, že *Command* je deaktivován, protože tlačítko nebo položka menu zešedne.

V každé WPF aplikaci existuje množství automaticky vytvořených *Commandů*, příkladem můžou být [14]:

- New (Nový)
- Save (Uložit)

- SaveAs (Uložit jako)
- Cut (Vyjmout)
- Copy (Kopírovat)
- Paste (Vložit)
- Open (Otevřít)
- Close (Zavřít)
- Print (Tisk)
- PrintPreview (Náhled tisku)
- Undo (Zpět)
- Redo (Znovu)
- A další

Mimo nich je možné vytvořit *Command* vlastní, odvozením z interface *ICommand*.

Před tím, než se *Command* připojí ke grafické komponentě, musí proběhnout tzv. *Command binding*, který specifikuje metodu, která se vykoná, když je *Command* aktivován a nepovinně také metodu pro zjištění, zda je *Command* aktivní. *Command binding* se zapisuje jako vnořený element k elementu, pro který *Command* definujeme, nejčastěji to bývá okno aplikace a má tvar *název_elementu.CommandBindings*.

```
<Window.CommandBindings>
      <CommandBinding Command="ApplicationCommands.Print"
                              Executed="PrintCommand_Executed"
                              CanExecute="CheckPrintableContent" />
</Window.CommandBindings>

<MenuItem Header="Tisk" Command="Print">
```

Obrázek 12: Command binding a použití Command

Command definovaný pomocí *bindingu* a následné použití u položky menu.

2.15 Distribuce aplikace

Existují tři možnosti, jak lze aplikace postavené na WPF distribuovat. První z nich je klasický zkompilovaný binární soubor s příponou *.exe*, který pro své spuštění nepotřebuje nic jiného, než operační systém Windows s nainstalovaným .NET Framework v příslušné verzi. Druhou je vytvoření aplikace typu XAML Browser Application s příponou *.xbap*. Takové aplikace ke svému spuštění potřebují ještě program Internet Explorer, ve kterém jsou hostovány. Takové aplikace můžou obsahovat jak kód XAML, tak i C# a jsou kompilovány, ale jsou limitovány určitými bezpečnostními

omezeními. Třetí možností je vytvoření samostatného XAML souboru s příponou *.xaml*, který je možné spustit v programu Internet Explorer. Takové aplikace ale nemohou obsahovat žádnou logiku v jazyce C# [5].

2.16 Uživatelské grafické komponenty

Mimo již předdefinovaných grafických komponent WPF umožňuje vytvoření vlastní komponenty s možností odvození od připravené základní třídy. Použitím odvození je programátorovi usnadněna práce, protože vytvářená komponenta již získá některé základní vlastnosti. Třídami, od kterých je vhodné odvozovat, jsou *UserControl* a *CustomControl*.

Při odvozování od *CustomControl* má vývojář více možností v implementaci, protože jsou odvozeny pouze základní vlastnosti, vytvoření vlastní komponenty odvozením od *CustomControl* je ale také podstatně náročnější.

Odvození od *UserControl* je na druhou stranu podstatně jednodušší, protože výsledek je pouze složen z více již existujících komponent. Na rozdíl od *CustomControl* chybí podpora skinů, to znamená, že vzhled se již po naprogramování nedá změnit [15].

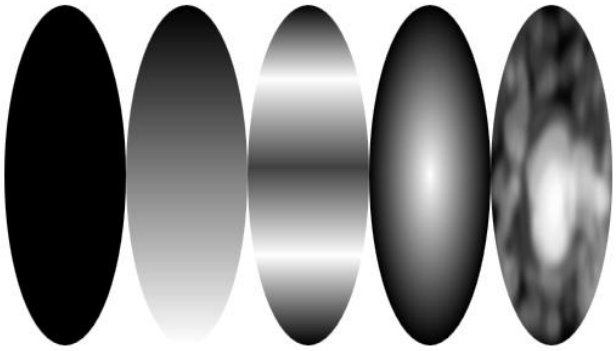
2.17 Štětce

Většina grafických komponent umožňuje upravit svůj vzhled definováním barev svých částí. Vlastnosti grafických komponent určujících barvu však nejsou definovány třídou *Color*, která ve WPF slouží k uložení barev, ale třídou *Brush*. *Brush*, neboli štětec je mnohem mocnější nástroj než samotná barva, protože umožňuje mimo barvy i aplikaci různých efektů, popřípadě textur [5]. Existují tyto třídy odvozené od třídy *Brush*:

- *GradientBrush*
  - *LinearGradientBrush*
  - *RadialGradientBrush*
- *SolidColorBrush*
- *TileBrush*
  - *DrawingBrush*
  - *ImageBrush*
  - *VisualBrush*

Nejjednodušším štětcem je *SolidColorBrush*, který představuje pouze konkrétní barvu. *LinearGradientBrush* a *RadialGradientBrush* slouží k tvorbě přechodů a jsou definovány více barvami. *LinearGradientBrush* je jednoduchý lineární přechod s možností nastavení orientace a rychlosti, případně poměru barev v přechodu. *RadialGradientBrush* obarví plochu tak, že vypočítá přechod mezi barvami pomocí elipsy, kterou lze parametricky definovat.

Třídy odvozené od *TileBrush* slouží k malování pomocí obrázků.



Obrázek 13: Použití štětců

Příklad dokumentuje použití štětců na elipse. Zleva *SolidColorBrush*, *LinearGradientBrush* se dvěma barvami, *LinearGradientBrush* s více barvami, *RadialGradientBrush* a *ImageBrush*.

3 Popis funkcí a prostředí aplikace

3.1 Rozšíření

V aplikaci byla implementována všechna navržená rozšíření z předchozí práce:

- Přidání třetí platformy aplikací s grafickým rozhraním.
- Stanovení typu výstupní platformy při vytváření nového projektu.
- Možnost tvorby strukturovaných datových typů (polí).
- Grafický simulátor vytvořených algoritmů.
- Umožnění tvorby libovolného množství algoritmů (funkcí) v rámci jednoho projektu.

Mimo těchto rozšíření byly také navrženy a implementovány některé funkce zlepšující uživatelský komfort při práci s programem.

3.1.1 Podpora přiblížení a oddálení pracovní plochy (zoom)

S tím, že GUI vývojového prostředí je vektorové, se váže jedna velká výhoda, jednoduchá implementace funkce zoom bez ztráty kvality zobrazení. Pro zoom jsou využívány grafické transformace, které jsou dostupné pro každou grafickou komponentu systému WPF. Pracovní plocha určená pro tvorbu diagramů je odvozena od grafické komponenty *Canvas*. *Canvas* je komponenta umožňující absolutní umísťování vnořených komponent, proto byla pro tento účel nejvhodnější. Samotná grafická transformace je pak implementována pomocí nastavení vlastnosti *RenterTransform*. Výsledek má tvar *canvas.RenterTransform = new ScaleTransform(scaleX,scaleY),* kde *scaleX* a *scaleY* určují výsledné zvětšení nebo zmenšení.

3.1.2 Podpora funkce schránky při tvorbě diagramů

Funkce schránky je implementována jak pro jednotlivé symboly, tak při kopírování více symbolů pro spojnice mezi kopírovanými symboly. Dále jsou, pokud to je možné, kopírovanému symbolu zachovány jeho definované parametry. Parametry ale musí být vymazány v případě, že je symbol přenášen mezi funkcemi. Důvodem je možná nekompatibilita lokálních proměnných. V takovém případě se vloží pouze kostra diagramu s prázdnými symboly.

3.1.3 Knihovna funkcí

Uživatel dostal možnost vytvořenou funkci exportovat do souboru a uložit ji do tzv. knihovny funkcí. Uložená funkce může být poté importována a používána v jiných projektech. Soubory exportovaných funkcí mají příponu *.vvdf* a formát XML. Struktura souboru je vytvořena jednoduchou serializací objektu funkce pomocí *DataContractSerializer*. Při exportu funkce bylo nutné řešit několik problémů. Prvním z nich byla práce s globálními proměnnými. Pokud funkce využívá nějakou globální proměnnou, pro zachování funkčnosti je nutné tuto proměnnou exportovat také. Problém

může nastat při následném importu. Při importu je nutné ověřovat, zda importovaná dodatečná globální proměnná nebude kolidovat s nějakou již existující. V případě kolize je nutné porovnat datové typy kolidujících proměnných, v případě odpovídajících typů se proměnné sloučí a v případě odlišných typů nebude import umožněn.

Speciální přístup vyžadoval případ, kdy exportovaná funkce ve svém těle volala nějakou další funkci. V tomto případě musela být zmíněná funkce exportována jako součást funkce hlavní a s globálními proměnnými vedlejší funkce bylo zacházeno obdobně.

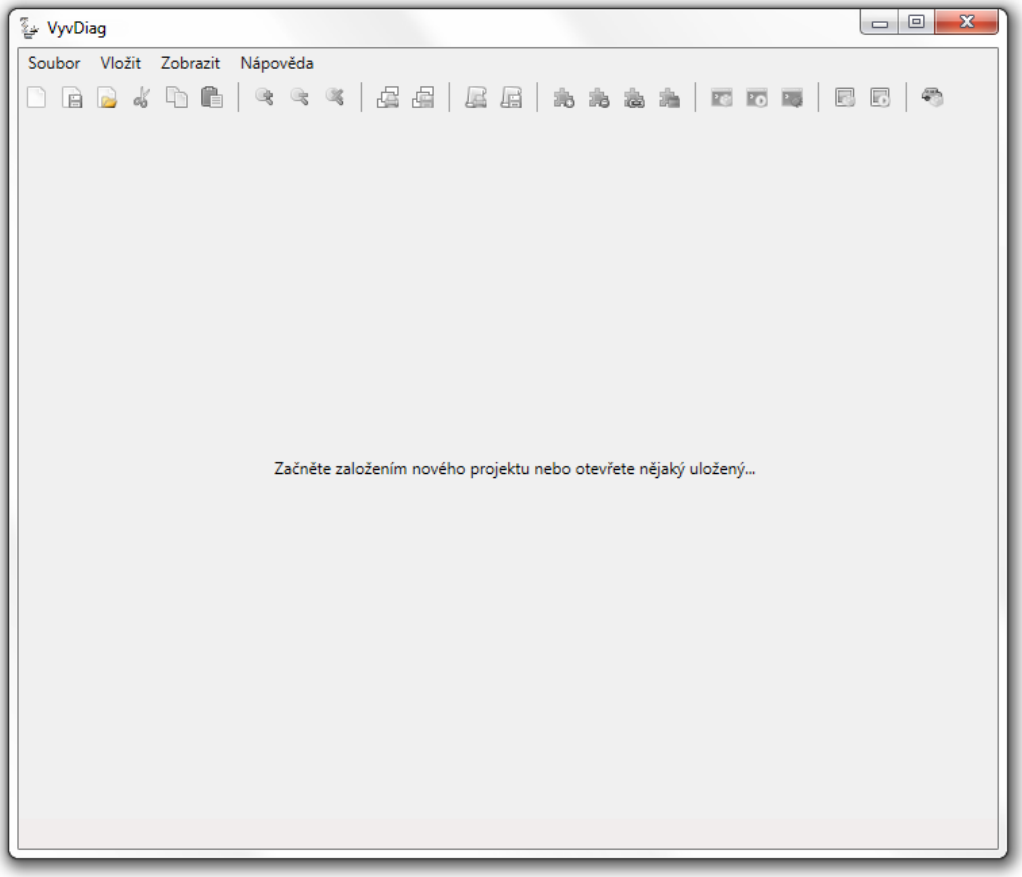
Export funkcí má také jasně definovaná omezení, není umožněno exportovat hlavní funkce programu (*main*), dále funkce ovlivňující grafické rozhraní, nebo některé pomocné proměnné pro práci s mikroprocesorem.

3.2 Prostředí

Následující kapitoly popisují prostředí aplikace. Rozšířený popis a funkčnost jsou popsány v manuálu aplikace, který je součástí této práce.

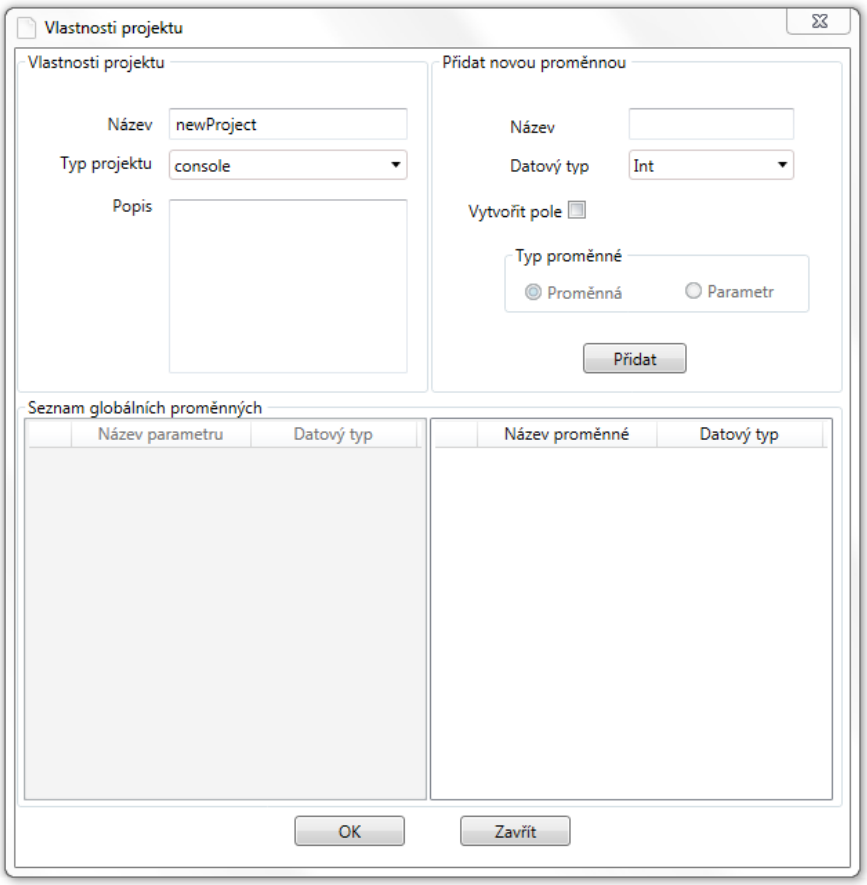
3.2.1 Tvorba aplikací

Po prvním spuštění aplikace se otevře hlavní okno aplikace s jednoduchou hláškou o absenci projektu, většina položek hlavního menu a nástrojové lišty není aktivní.



Obrázek 14: Hlavní okno aplikace

Prvním logickým krokem je vytvoření nového projektu, čehož může uživatel docílit z hlavního menu položkou Soubor – Nový projekt, nebo ikonou bílé stránky z hlavního panelu, případně klávesovou zkratkou *Control+N*. Tato akce vyvolá okno vlastností projektu.

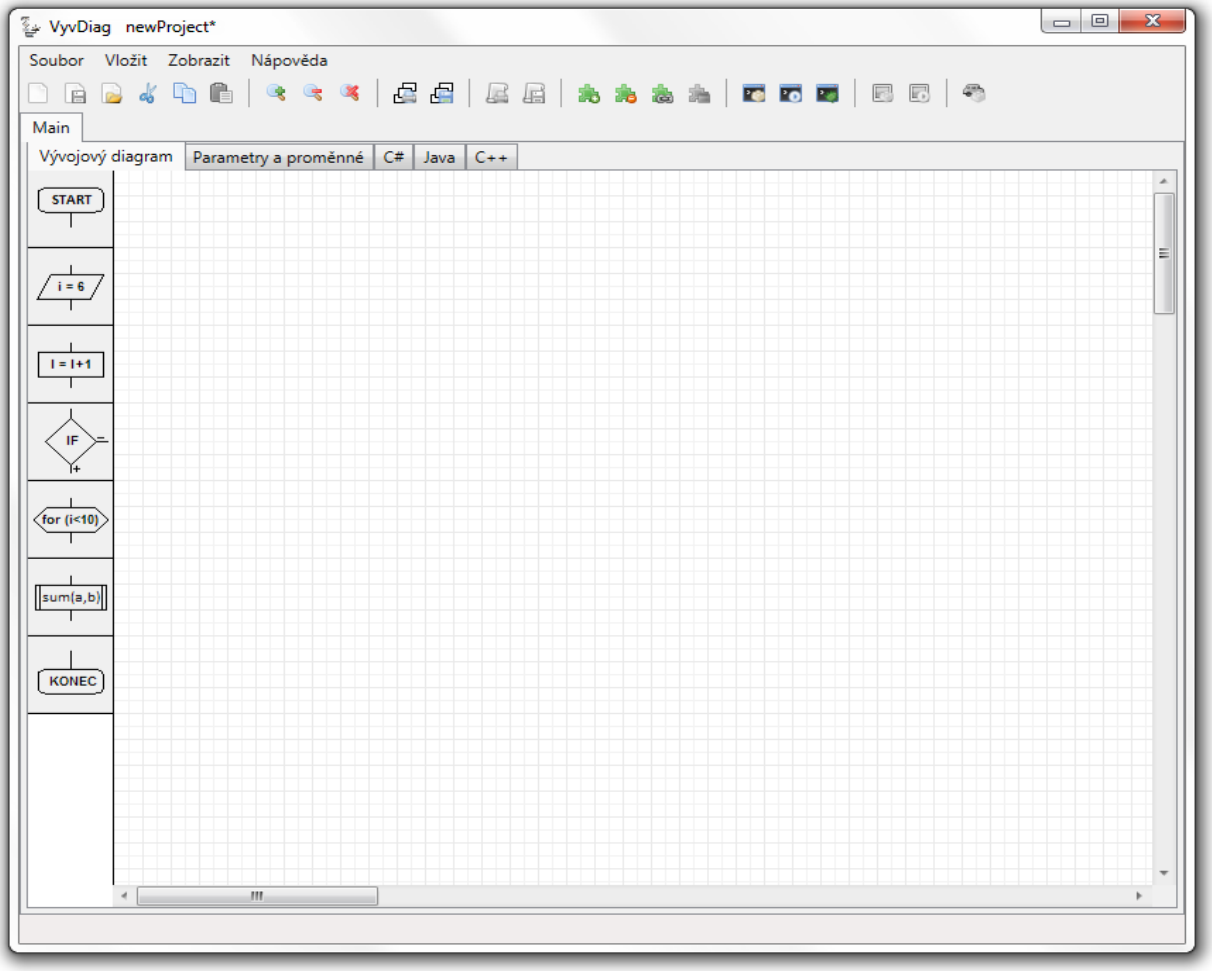


Obrázek 15: Okno vlastností projektu

Okno je rozděleno na tři hlavní částí. Vlevo nahoře se nachází box s vlastnostmi projektu, uživatel může vyplnit název projektu, který je použit i pro jména následně vytvořených souborů, proto je omezen na znaky anglické abecedy bez mezer a čísla, přičemž číslem nemůže začínat. Dále typ projektu, kterým může být konzolová aplikace, aplikace s grafickým rozhraním, nebo aplikace pro mikroprocesor. Třetím polem je pole pro popis projektu. Popis se poté zobrazí také ve vygenerovaných zdrojových kódech. Vpravo nahoře se nachází část pro přidávání globálních proměnných projektu, které jsou dostupné ve všech vytvořených funkcích. Při vytváření proměnné je nutné vyplnit její název, pro něhož platí klasická pravidla tvorby názvů proměnných ve vyšších programovacích jazycích, datový typ, kterým může být *integer*, *double*, *string*, *boolean*, nebo *char*, volitelná je možnost vytvoření pole, při jejímž zvolení je nutné zadat počet prvků pole. V dolní části okna je seznam vytvořených globálních proměnných a ovládací prvky pro zavření okna.

Po vyplnění informací o projektu, případně vytvoření globálních proměnných, je projekt otevřen v hlavním okně. Automaticky je u každého typu projektu vytvořena hlavní metoda, zpravidla

pojmenovaná *Main* a aktivují se některé položky menu a ikony hlavního panelu dle zvoleného typu projektu.



Obrázek 16: Prostředí s vytvořeným projektem

V nejhornější části se nachází titulek okna obsahující název programu a projektu, příznak neuložených změn (*) a pokud proběhlo uložení, tak cesta k uloženému souboru.

Pod titulkem okna se nachází hlavní menu programu obsahující většinu funkcí programu. Jeho struktura je následující:

- Soubor
  - Nový
  - Otevřít
  - Uložit
  - Uložit jako
  - Tisk
  - Zavřít projekt
  - Konec
- Vložit

    - Funkci
    - Symbol
        - Start
        - Konec
        - Podmínka
        - Cyklus
        - Funkce
        - Vstup/Výstup
        - Zpracování
    - Grafickou komponentu
        - Popisek
        - Tlačítko
        - Zaškrtávací tlačítko
        - Výběrové tlačítko
        - Ukazatel průběhu
        - Vstup textu
- Zobrazit
    - Vlastnosti projektu
    - Knihovnu funkcí
    - Výběr jazyka rozhraní
- Nápověda
    - Zobrazit nápovědu
    - Klávesové zkratky
    - O programu

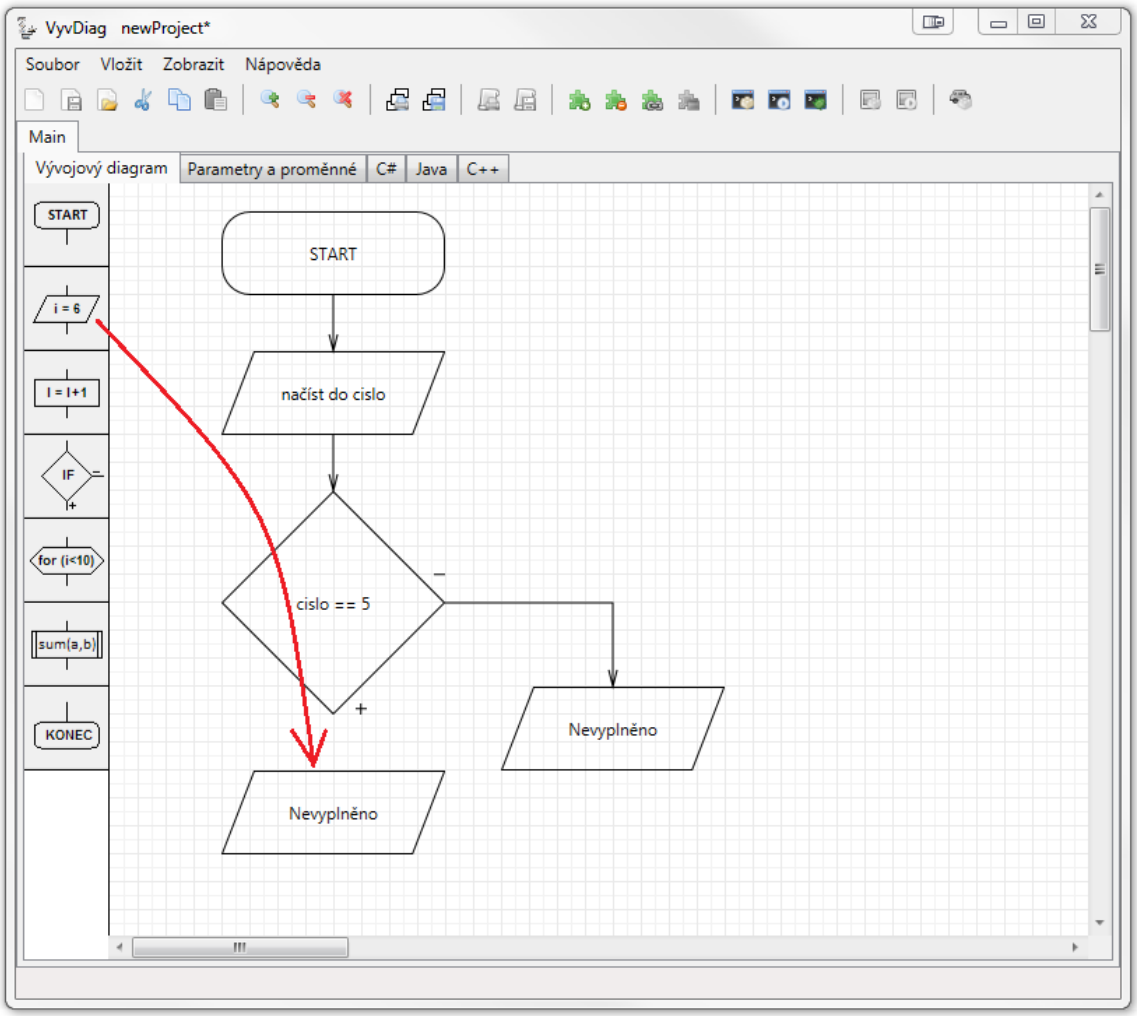
Pod hlavním menu je nástrojová lišta obsahující nejpoužívanější funkce. Lišta je rozdělena na části oddělené oddělovačem. V první části jsou ikony pro práci s projektem – vytvoření nového, uložení a otevření a také se schránkou – kopírovat, vyjmout a vložit. V další části jsou ikony lupy pro přiblížení, oddálení a navrácení na implicitní přiblížení. Následují ikony pro tisk a export vytvořených diagramů, tisk a export vygenerovaných zdrojových kódů. Sada čtyř ikon se zeleným dílkem puzzle slouží ke správě funkcí – přidání nové, odebrání aktuální, kontrola validity a export do knihovny. Poslední tři oddíly slouží každý pro jeden typ projektů. První pro konzolové projekty a umožňuje jejich kompilaci, spuštění a simulaci, druhý pro grafické projekty umožňující kompilaci a spuštění a poslední pro projekty pro mikroprocesor umožňující pouze kompilaci.

Hlavnímu oknu dominuje komponenta se dvěma řadami záložek. Každá vytvořená funkce obohatí hlavní komponentu o jednu záložku obsahující 3 až 5 záložek vnořených. Počet vnořených záložek je definován typem otevřeného projektu. U konzolových aplikací jich je 5 a to:

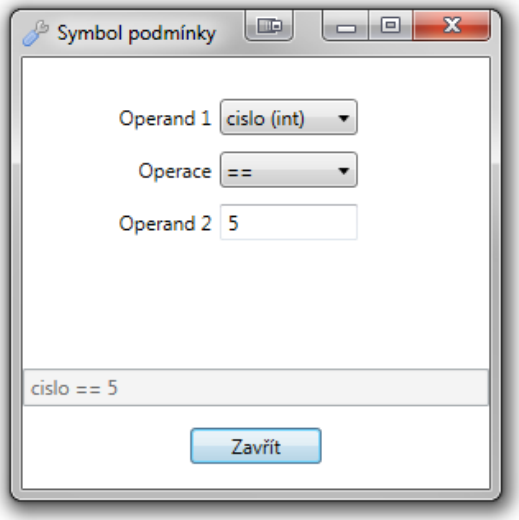
- Pracovní plocha pro tvorbu diagramů
- Formulář pro nastavení vlastností funkce, tvorbu lokálních proměnných a parametrů
- Vygenerovaný zdrojový kód v jazyce C#
- Vygenerovaný zdrojový kód v jazyce Java
- Vygenerovaný zdrojový kód v jazyce C++

U grafických aplikací přibude záložka s editorem grafického rozhraní a zdrojový kód bude generován pouze v jazyce C# a XAML. U aplikací pro mikroprocesor to bude pouze jazyk C.

Pracovní plocha pro tvorbu diagramů je rozdělena na dvě části, vlevo se nachází lišta s dostupnými grafickými symboly, ze kterých se diagram sestavuje, a vpravo je plocha pro sestavovaný diagram. Samotné sestavování se skládá z několika částí, uživatel metodou *drag and drop* přetáhne potřebné symboly na pracovní plochu, každý symbol má spojovací body, ve kterých začínají nebo končí spojnice vedoucí do ostatních symbolů. Každý symbol má minimálně jeden spojovací bod. Spojovací body se nachází ve středech hran symbolů. Aby diagram byl validní, každý spojovací bod každého symbolu musí obsahovat spojnici. Z toho vyplývá, že dalším bodem tvorby diagramu je spojování symbolů. Následuje vyplnění parametrů symbolů, případně vytvoření potřebných proměnných či parametrů funkce. Vyplňování parametrů se provádí ve speciálních oknech, která se vyvolají po dvojkliku na zvolený symbol, nebo označením symbolu a stiskem klávesy *Enter*.



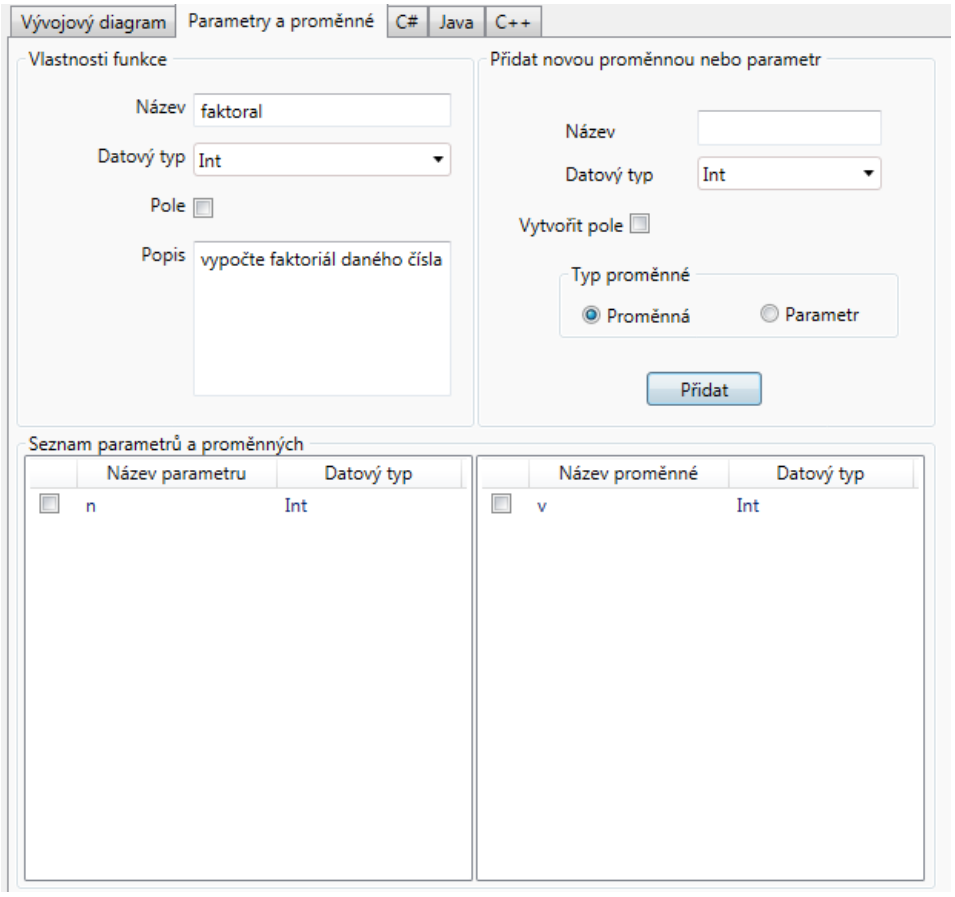
Obrázek 17: Tvorba diagramu



Obrázek 18: Okno parametrů symbolu podmínky

Okno pro zadávání parametrů symbolů obsahuje mimo jiné také pruh s textovým popisem obsahu.

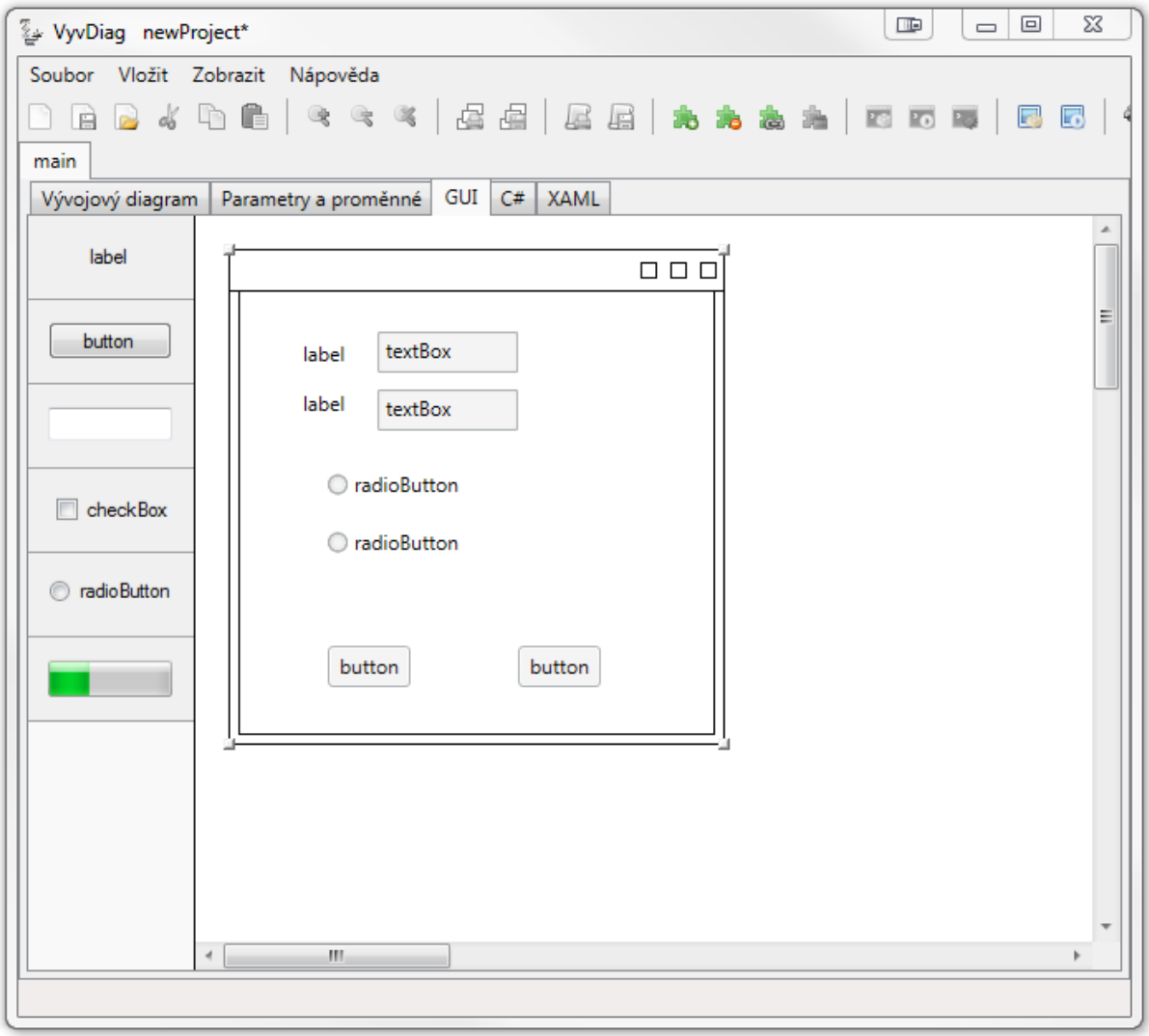
Formulář pro nastavení vlastností funkce má podobné rozhraní jako okno vlastností projektu. Rozdílem je možnost vyplnit datový typ návratové hodnoty funkce a je aktivní seznam parametrů.



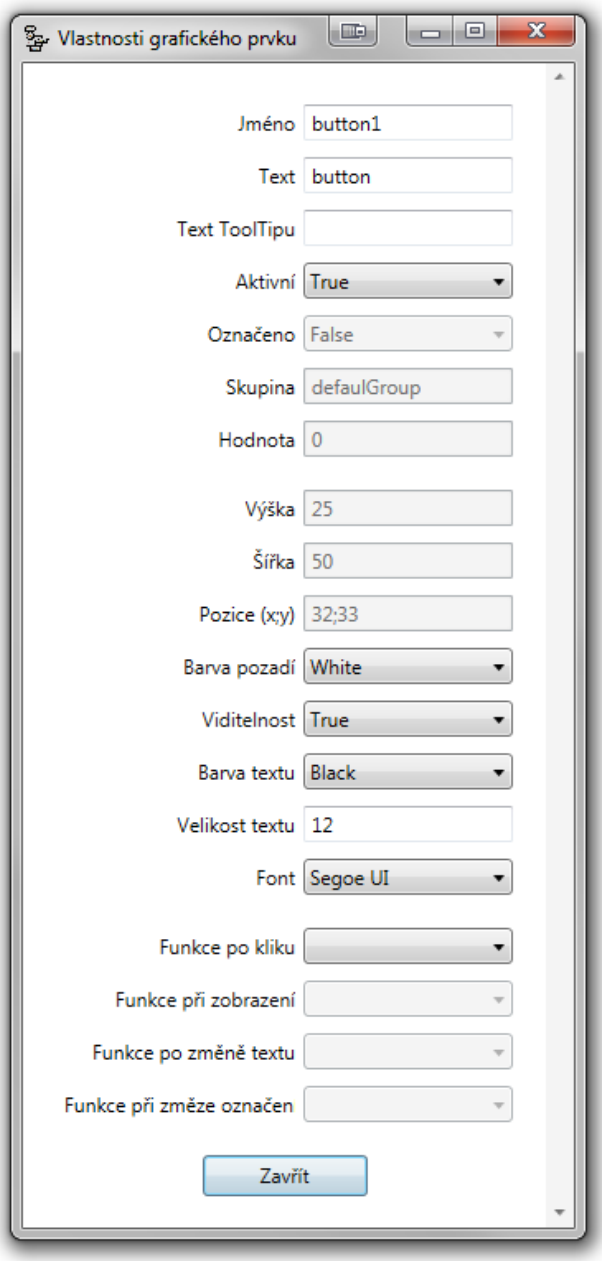
Obrázek 19: Nastavení vlastností funkce, tvorba lokálních proměnných a parametrů

Následující záložky obsahují pouze texty vygenerovaných zdrojových kódů s barevně zvýrazněnou syntaxí. Poslední záložkou, která se v liště vyskytne, pouze když je otevřen projekt s aplikací s grafickým rozhraním, je editor grafického rozhraní.

Editor grafického rozhraní je rozdělen na dvě části obdobně jako editor vývojových diagramů. Vlevo se nachází seznam dostupných grafických komponent, které se do tvořeného rozhraní vkládají metodou *drag and drop*. Vpravo je pracovní plocha. Prázdná pracovní plocha obsahuje pouze prázdné okno tvořeného programu, do kterého se vkládají další komponenty. Uživatel má možnost měnit pozici a rozměry všech komponent. Poklepáním myši na konkrétní komponentu se vyvolává formulář pro zadávání jejich vlastností, kde některé položky nemusí být aktivní, aktivita se řídí typem zvoleného prvku.



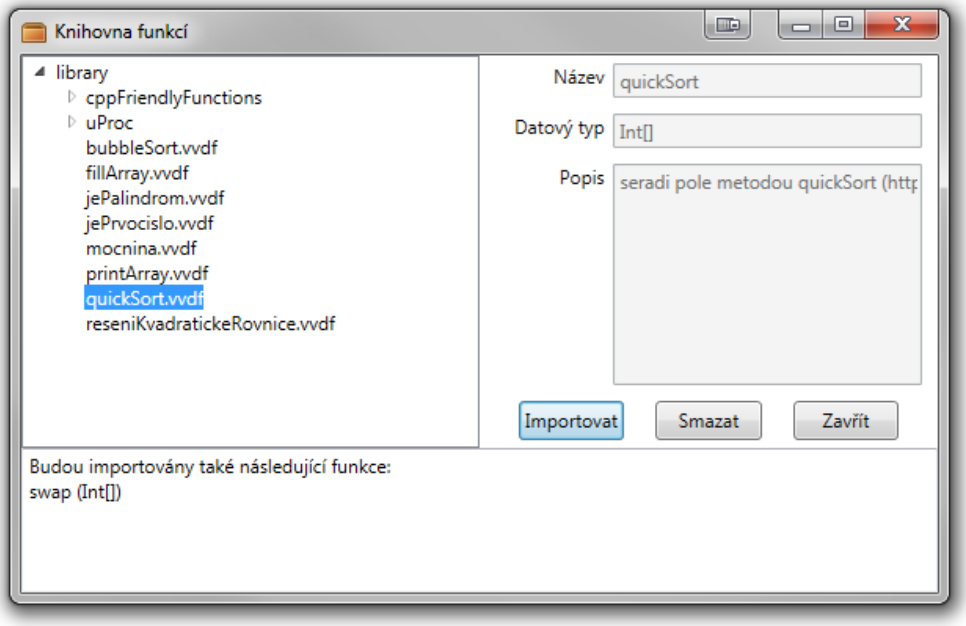
Obrázek 20: Editor grafického rozhraní



Obrázek 21: Okno vlastností grafického prvku

3.2.2 Knihovna funkcí

Uživatel má při správě funkcí možnost využít knihovny funkcí. Knihovna je vyvolána z hlavního menu a její rozhraní se otevře v novém okně.



Obrázek 22: Knihovna funkcí

Rozhraní knihovny je jednoduché, vlevo je struktura knihovny, která přesně odpovídá adresářové struktuře v umístění instalace programu. Po zvolení funkce se zobrazí její podrobnosti a v dolní části také podrobnosti importu – závislé funkce, importované se zvolenou funkcí a případně používané globální proměnné.

3.2.3 Simulátor

Simulátor algoritmů má své rozhraní rovněž ve speciálním okně, které je dostupné ve dvou režimech, které se od sebe liší pouze dostupnými funkcemi.

Simulátor ve své horní části obsahuje nástrojovou lištu, kterou se ovládá. Obsah této lišty se mění dle zvoleného režimu. Při jednoduchém režimu jsou dostupné jen základní příkazy

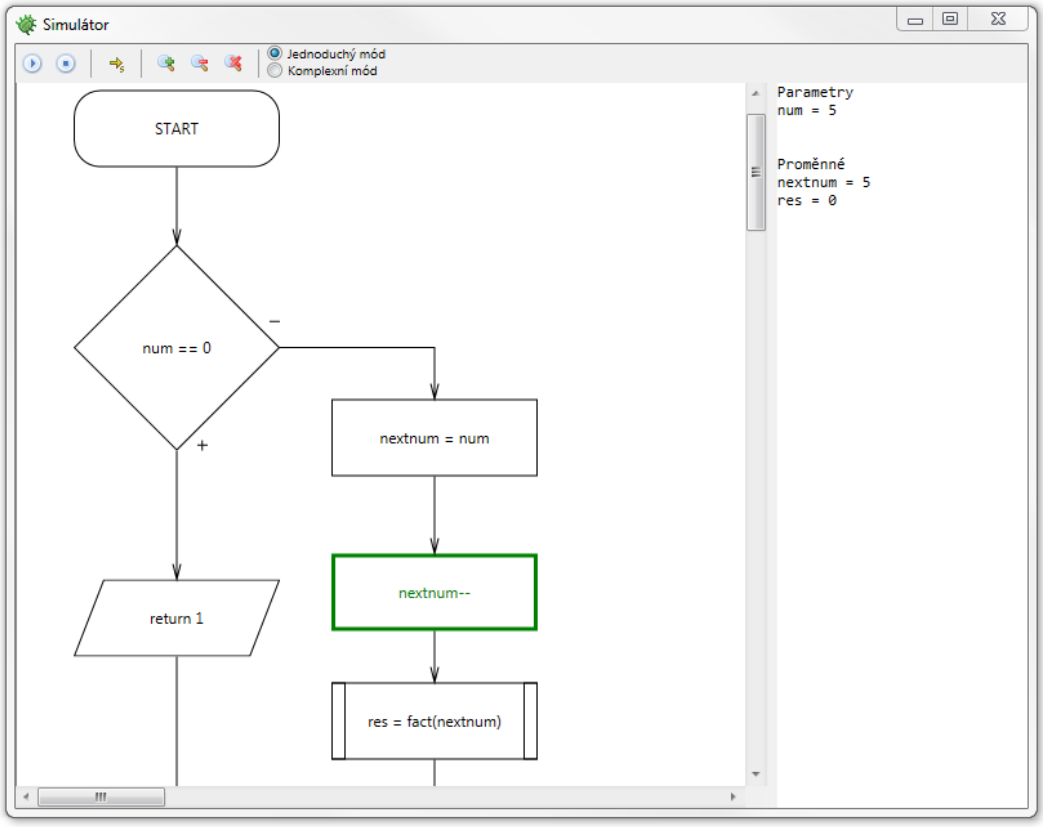
- Zahájení a ukončení simulace
- Skok na další symbol algoritmu
- Funkce přiblížení

Zvolením komplexního módu se příkazy rozšíří o

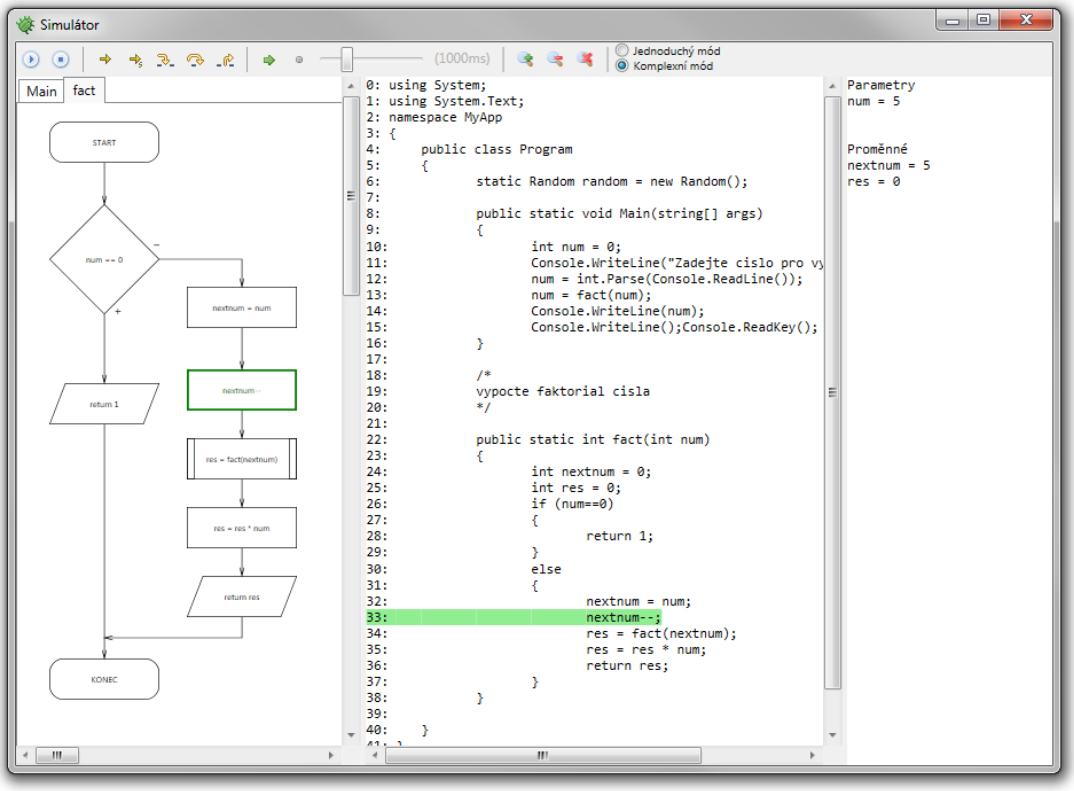
- Krokování po řádcích zdrojového kódu
- Skoky do, přes a ven z funkcí
- Automatické krokování po řádcích kódu v čase s možností nastavení intervalu od 300 ms do 3 s

Vzhled spodní části je také ovlivněn zvoleným módem. V jednoduchém režim jsou v levé části na záložkách zobrazeny jednotlivé vývojové diagramy funkcí se zeleně označeným aktuálně zpracovávaným symbolem a v pravé části se po spuštění simulace zobrazují stavy proměnných.

Přepnutím do komplexního režimu se prostředí rozšíří o třetí část, zobrazený zdrojový kód, aktuální řádek je označen zeleným podbarvením.



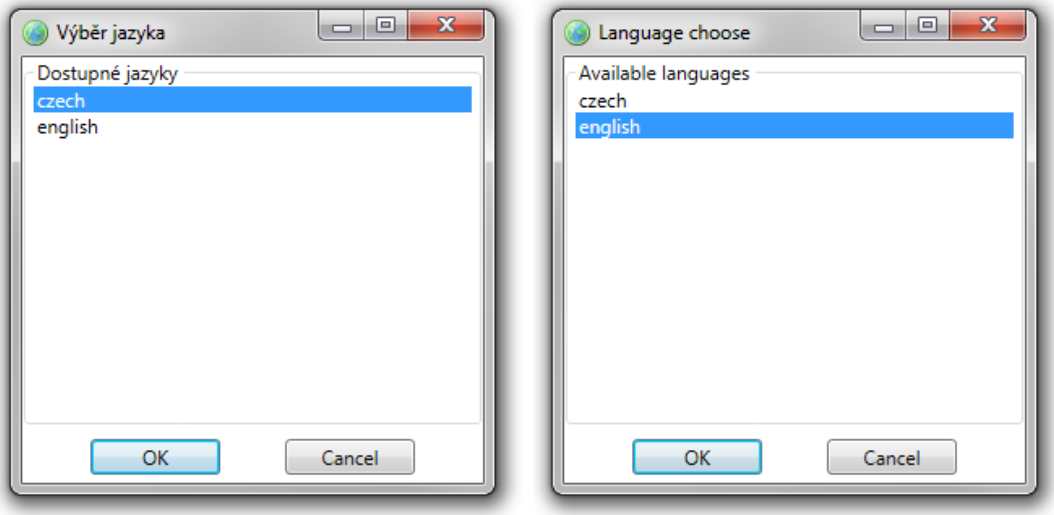
Obrázek 23: Okno simulátoru v jednoduchém režimu



Obrázek 24: Okno simulátoru v komplexním režimu

3.2.4 Změna jazyka

Aplikace umožňuje dynamickou změnu jazyka rozhraní bez nutnosti restartu aplikace. Momentálně jsou dostupné dva jazyka a to čeština a angličtina.



Obrázek 25: Rozhraní pro změnu jazyka

4 Implementační detaily

S přepsáním celé aplikace a použitím nové knihovny Windows Presentation Foundation se váží i změny některých základních implementačních principů, hlavně v tvorbě diagramů a několik zajímavých implementačních detailů.

4.1 Princip aplikace

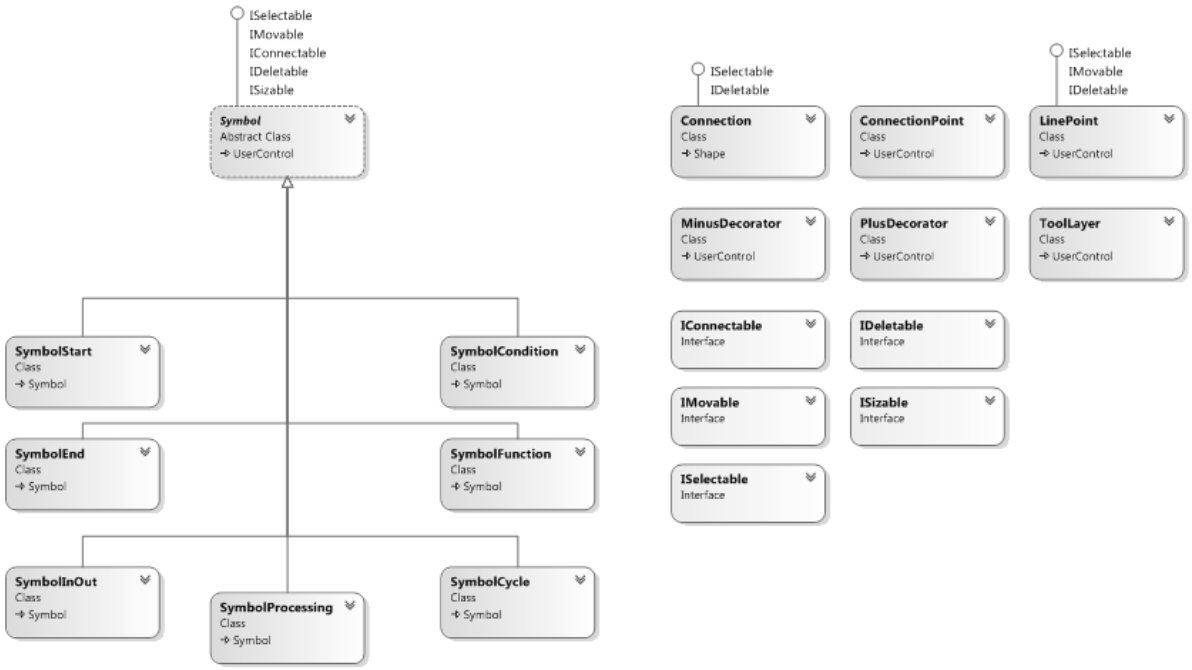
Propojení logiky aplikace s jejím grafickým rozhraním je docíleno použitím příkazů (*Command).* Hlavní třída aplikace obsahuje z větší části pouze metody vyvolané aktivací příkazů, případně metody pro testování aktivity příkazu. Kód obsluhující příkazy je rozdělen na několik regionů

- *File Commands* – pro klasické příkazy společné většině aplikací (Nový projekt, uložit, tisk atd.)
- *Cut, copy, paste* – pro obsluhu schránky
- *Other commands* – pro pomocné příkazy, jako jsou například vložení nového symbolu do diagramu, otevírání nápovědy, okna jazyků a další
- *Zoom commands* – pro funkce přiblížení či oddálení diagramů, nebo změnu velikosti písma kódu
- *Printing commands* – pro obsluhu tisku
- *Function commands* – pro obsluhu práce s funkcemi
- *Code commands* – pro práci se zdrojovým kódem
- *Diagram commands* – pro práci s diagramem
- *Console project commands* – pro práci s konzolovými projekty
- *Graphics project commands* – pro práci s projekty s grafickým rozhraním
- *uProc project commands* – pro práci s projekty pro mikroprocesor
- *Command checkers* – metody ověřující aktivitu příkazů

4.2 Tvorba diagramů

Princip grafické tvorby diagramů byl zcela přepracován. Původně bylo nutné všechny funkce, od testu kliku myší na symbol, výpočtu vzhledu symbolů, či přepočítávání rozměrů implementovat ručně, takže logika zajišťující tuto funkčnost byla velice rozsáhlá. S použitím WPF většina těchto problémů odpadá.

Všechny prvky diagramu, ať už se jedná o symboly, nebo spojnice, jsou odvozeny od základních tříd grafických komponent systému WPF, viz následující diagram.



Obrázek 26: Struktura tříd grafických prvků

Základní třídou pro všechny symboly je třída *Symbol* odvozená od třídy *UserControl*. Ve třídě *Symbol* je definována logika společná všem symbolům a to například podpora přesouvání a změny rozměrů myší, indikace, zda je symbol uživatelem vybrán, protože vybrané symboly mají svou grafiku vykreslenou silnějšími čarami, dále je definována podpora *ToolTip*, což je jednoduchá kontextová nápověda, vyvolaná najetí myší. Kostra většiny funkcí je definována rozhraními. V třídě Symbol jsou implementována tato rozhraní:

- *ISelectable* – definuje možnost uživatelského vybrání symbolu
- *IMovable* – posouvání symbolu myší
- *IConnectable* – spojování symbolu myší, uchovává pozice spojovacích bodů a reference na připojené spojnice
- *IDeletable* – smazání symbolu
- *IScalable* – změna rozměrů

Od třídy *Symbol* jsou odvozeny třídy pro konkrétní symboly. Odvozené třídy většinou nepřidávají žádnou funkčnost, ale pouze definují grafiku symbolu. Grafika se skládá z několika částí, nejhlavnější je vlastní tvar symbolu. Tvar je složen ze základních grafických primitiv, jako jsou obdélníky, kruhové výseče, nebo samostatné čáry. Velkou výhodou použití základních primitiv je možnost definovat automatické reakce na vlastnosti objektu pomocí stylů. Například lze nastavit automatickou reakci na změnu příznaku aktivity symbolu spočívající ve změně tloušťky čar. Tohoto chování je docíleno pomocí stylování popsaného v kapitole popisu vlastností WPF.

Mimo tvaru grafika symbolu obsahuje ještě několik dalších částí. Je to textové pole popisující vnitřní stav symbolu, množina spojovacích bodů, z kterých a do kterých vedou spojnice a tzv. *ToolLayer*. Tato vrstva obsahuje prvky, jež slouží ke změně rozměrů symbolu. I tyto prvky jsou součástí základní sady grafických komponent a nazývají se *Thumb*, uživatel si jich může všimnout při označení symbolu a při přesunu myši nad jejich grafiku se změní tvar kurzoru dle dostupného směru změny velikosti.

Grafiku diagramu tvoří mimo symbolů také spojnice. Spojnici symbolizuje třída *Connection* odvozená od třídy *Shape*. Třída *Connection*, stejně jako třída *Symbol* definuje reakce na akce myši a dále vlastnosti specifické pro spojnice. Uchovává referenci na body, ve kterých začíná a končí, a které jsou zároveň pozicemi spojovacích bodů symbolů, které spojnice spojuje. Pokud je tvar spojnice složitější, udržuje třída také seznam tzv. kolen, tedy bodů, kde je spojnice zalomená. Na těch bodech zalomení jsou umístěny také speciální komponenty definované třídou *LinePoint*, které zajišťují možnost zalomení přesouvat a mazat pomocí myši.

První, poslední a body zalomení určují geometrii vykreslované čáry představující spojnici. Při vykreslování se ale projevil první nedostatek WPF oproti starší a dříve použité knihovně Windows Forms. WPF neumožňuje automaticky vykreslit na konec čáry šipku. Geometrie šipky, skládající se ze dvou přidaných čar, musí být ručně dopočítána.

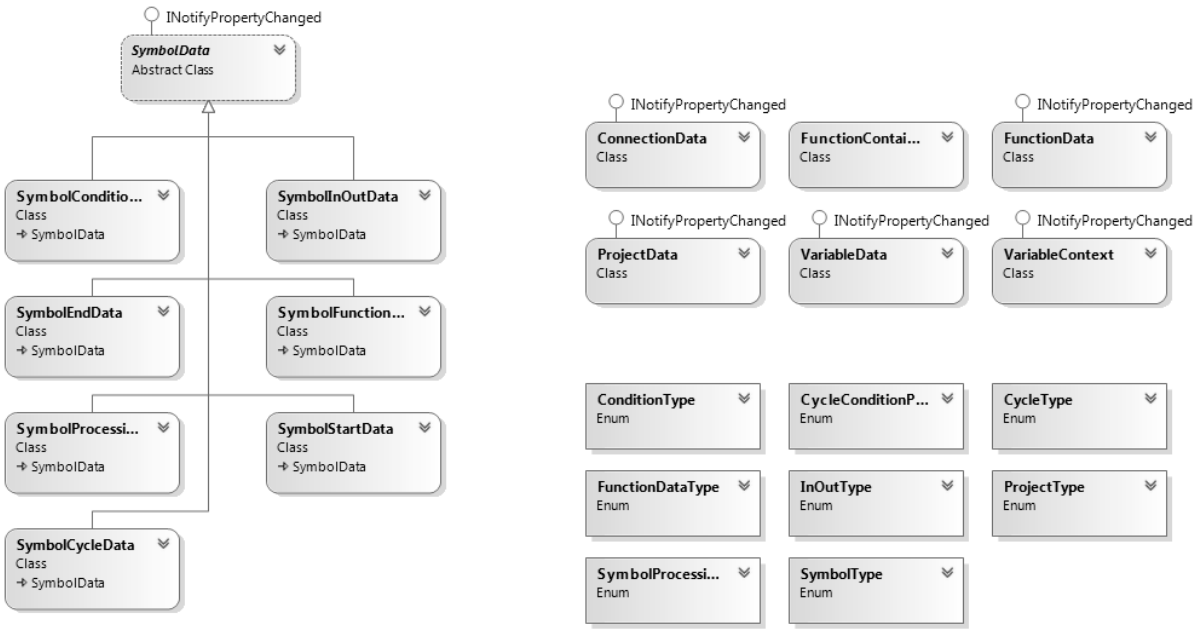
Popsané grafické prvky tvoří grafiku diagramu, která ale musí být někde umístěna. Jako plátno či pracovní plocha slouží třída *DiagramCanvas* odvozená od třídy *Canvas*. *Canvas* je komponenta typu *ContentControl*, je odvozená od třídy *Panel* a má vlastnost *Children*. *Children* je množina vnořených komponent, které tato komponenta spravuje. *Canvas* pro definici pozic svých vnořených komponent používá absolutní souřadnice. Tyto souřadnice nejsou uloženy ve vlastnostech jednotlivých vnořených komponent, ale určení pozice se provádí speciálními metodami *SetLeft(vzdálenost)*, *SetTop(vzdálenost)*, kde parametr určuje odstup komponenty od levého/horního okraje *Canvas*.

Třída *DiagramCanvas* rozšiřuje *Canvas* o všechny potřebné funkce pro tvorbu vývojových diagramů. Obsahuje metodu, která vygeneruje dříve zmíněné mřížky pro usnadnění zarovnávání symbolů k osám. Generování spočívá pouze v tom, že do vlastnosti *Children* se přidají čáry na definovaných pozicích. Dále definuje podporu pro výběr více symbolů myší zároveň. Metodu pro zpracování příkazů *drag and drop* pro přidávání nových symbolů, metody pro odstraňování symbolů a spojnic, které není úplně triviální. Při mazání symbolu je nutné ošetřit, aby se smazaly i spojnice tomuto symbolu náležející a pokud se spojnice skládá z více napojených částí, aby i tyto napojené části byly správně odstraněny.

Velkou část kódu tvoří metody pro spojování symbolů. Bylo nutné zajistit průběžné vykreslování tvořené spojnice, správné vytvoření objektu *Connection* a naplnění všech referencí nutných pro jeho fungování, v případě zrušení spojování správné vyčištění referencí na zrušenou spojnici atd.

4.3 Datové třídy diagramu

Třídy popsané v předchozí kapitole tvořily grafickou část diagramu. Grafika je ale jen visuální zobrazení dat, které jsou uloženy odděleně ve vlastních třídách. Tyto třídy jsou velice podobné třídám pro uchovávání dat z předchozí práce.



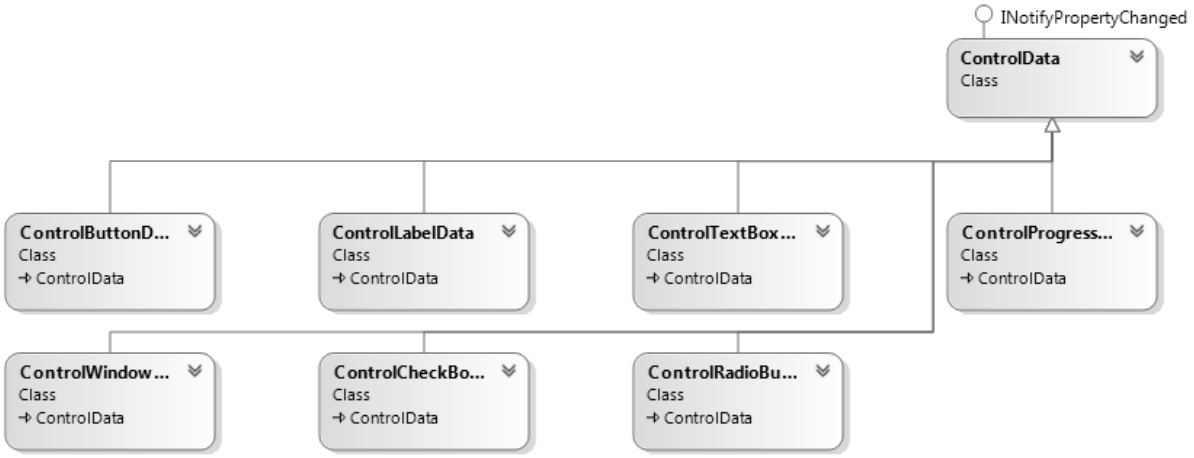
Obrázek 27: Struktura tříd pro uchovávání dat diagramu

Všechny datové třídy implementují rozhraní *INotifyPropertyChanged.* Implementací tohoto rozhraní je zajištěno upozornění grafických tříd, že se vnitřní data změnila a je nutné provést překreslení. Tento přístup se jmenuje *DataBinding* a byl popsán již dříve.

Bylo vytvořeno několik typů datových tříd. Třída *SymbolData* sloužící jako základ pro odvození datových tříd jednotlivých symbolů diagramu, *ConnectionData* uchovávající data spojnic, *FunctionData*,uchovávající data celých funkcí, *ProjectData* pro data projektů, *VariableData* pro data proměnných a *VariableContext* představující použití konkrétní proměnné v symbolu.

4.4 Tvorba grafického rozhraní aplikací

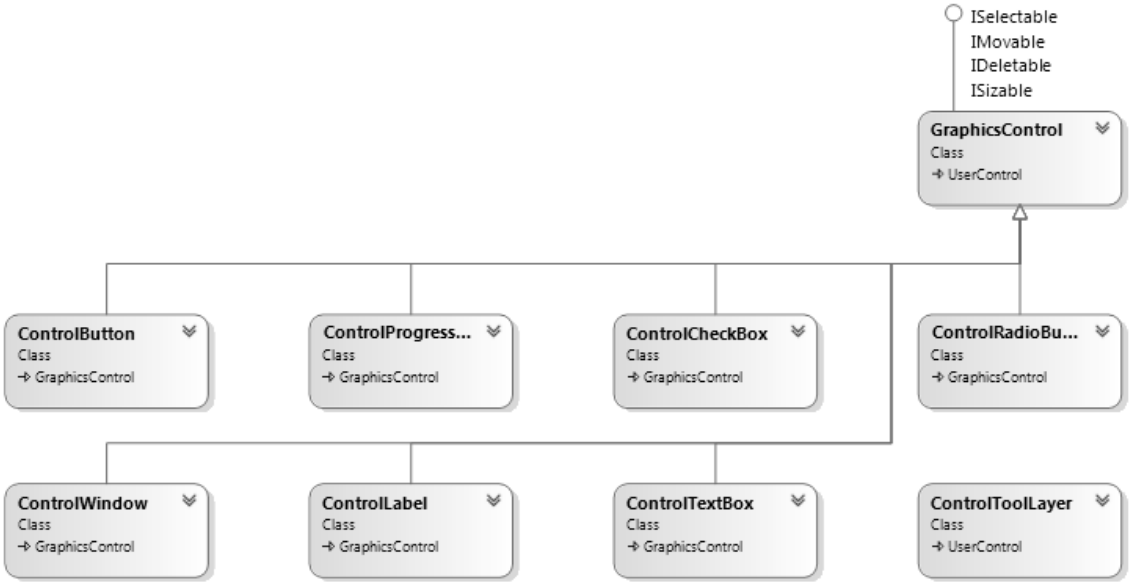
Implementace možnosti tvorby aplikací s grafickým rozhraním spočívala v několika krocích. Bylo nutné navrhnout strukturu tříd pro uchovávání dat jednotlivých grafických komponent použitých v rozhraní.



Obrázek 28: Struktura tříd pro uložení dat grafického rozhraní

Byla vytvořena jednoduchá základní třída *ControlData* uchovávající data, která jsou společná všem grafickým komponentám a od ní byly odvozeny třídy pro specifická data konkrétních komponent.

Druhým krokem byl návrh editoru grafického rozhraní. Princip editoru je velmi podobný editoru vývojových diagramů. Byla navrhnuta základní třída *GraphicsConrol* odvozená od třídy *CustomControl*, která definuje základní chování všech grafických komponent. Od této třídy byly odvozeny samostatné třídy pro každou komponentu. Tyto třídy definují vzhled a jsou podstatně jednodušší než třídy definující vzhled symbolů diagramu. Základem je vždy již vytvořená klasická grafická komponenta WPF, například u třídy *ControlButton* je to tlačítko, nad ní je jen vrstva umožňující změnu rozměrů a přesun.



Obrázek 29:Struktura tříd grafických komponent

Pracovní plocha editoru grafiky je tvořena třídou *GraphicsCanvas*, jež je založena na obdobném principu jako byla třída *DiagramCanvas*.

Třetím krokem bylo zpracování navrženého grafického rozhraní a vygenerování zdrojového kódu. Pochopitelným krokem bylo zvolení grafické knihovny WPF, takže se rozhraní zpracuje do kódu v jazyce XAML. K návrhu grafických rozhraní bylo definováno několik omezení pro co největší zjednodušení. Například jednotlivé grafické prvky nebudou umisťovány do layoutů, ale absolutně nastavovány na souřadnice. Z toho plyne, že prvním elementem vnořeným do vygenerovaného elementu *Window* bude *Canvas*. Dále jsou podstatně omezeny vlastnosti, které se dají grafickým prvkům definovat:

- Jméno sloužící jako jednoznačný identifikátor
- Text pokud komponenta nějaký zobrazuje
- *ToolTip* jako kontextová nápověda
- Aktivita překládaná jako vlastnost *IsEnabled*
- Označení u zaškrtávacích nebo přepínacích tlačítek
- Skupina pro komponentu *RadioButton*
- Hodnota pro komponentu *ProgressBar*
- Výška a šířka
- Pozice
- Barva pozadí a textu
- Viditelnost
- Velikost textu
- Font
- Metody volané jako reakce na události

- Po kliku
- Při zobrazení
- Po změně textu
- Po změně označení

Z vytvořeného grafického rozhraní se vygeneruje kód v jazyce XAML. Jazyk XAML je založen na XML, takže bylo vhodné použít zabudovaného generátoru XML dokumentů platformy .NET ze jmenného prostoru *System.XML.Linq*. Generování probíhá v daném pořadí, jako první je vždy zpracováno okno aplikace a poté jednotlivé komponenty. Generován je lehce neúsporný kód, protože jsou zpracovávány a nastavovány všechny parametry komponent, nehledě na implicitní hodnoty, takže i v případě, že některá vlastnost nebyla změněna a je na své implicitní hodnotě, nastavení této hodnoty bude i přesto zapsáno. Výsledkem generování je objekt typu *System.XML.Linq.XElement*.

4.5 Generování zdrojového kódu

Oblast, která prodělala jen menší změny, je generování zdrojových kódů. S umožněním tvorby funkcí byla původní třída generující zdrojový kód rozdělena na dvě, *FunctionCodeGenerator* pro zpracování jednotlivých funkcí a *ProjectCodeGenerator* pro zpracování celého projektu.

FunctionCodeGenerator při zpracování diagramu využívá stále stejný postup, z vývojového diagramu je vytvořen strom symbolů, který je poté procházen, a z jednotlivých symbolů jsou generovány řádky zdrojového kódu. Při zpracovávání stromu symbolů je také tvořen slovník pozic jednotlivých symbolů ve zdrojovém kódu, který je dále použit při simulaci pro označování symbolů. Třída musela být také upravena, protože byla rozšířena množina funkcí některých symbolů.

ProjectCodeGenerator je nová třída, která zajišťuje zpracování celého projektu. Zpracování probíhá v několika krocích. Je nutné vygenerovat úvodní části zdrojových kódů, obsahující pro jazyk C# příkazy *using*, definici jmenného prostoru, definici třídy programu a v případě grafického rozhraní konstruktor s voláním metody *InitializeComponent()*, která inicializuje grafické rozhraní. V případě jazyku Java je to deklarace *package*, importů, třídy a *Scanneru*, který slouží ke čtení z konzole. V případě jazyků C a C++ jsou to direktivy *#include*, definice makra pro práci s řetězci a použití jmenného prostou.

Za úvodní částí je v případě jazyků C a C++ nutné vygenerovat funkční prototypy. Poté jsou zpracovány všechny globální proměnné a jejich nastavení na implicitní hodnoty. Po globálních proměnných jsou zpracovány samotné funkce.

Pokud je generován kód pro projekt s grafickým rozhraním, po funkcích jsou zpracovány ještě události grafických komponent, do kterých jsou přiřazovány obslužné funkce. Třída *ProjectCodeGenerator* dále obsahuje metodu pro vygenerování zdrojového kódu definujícího grafické rozhraní u projektu s grafickým rozhraním.

4.6 Simulátor

Implementace simulátoru byla nejsložitějším novým prvkem aplikace. Simulátor samotný je složen z několika vrstev, které spolu komunikují.

Nejnižší vrstvu a jádro tvoří knihovna *Mdbg*. *Mdbg* je nadstavbou nad nízko úrovňovým ladícím API .NET Frameworku *ICorDebug*. Knihovna je složena z několika souborů: *mdbgeng.dll, NativeDebugWrappers.dll, raw.dll, corapi.dll*. Nevýhodou této knihovny je špatná dokumentace, ale existuje hotové *open source* řešení, které je nad touto knihovnou postaveno a které funkčnost nastiňuje velice dobře. Tato aplikace má název *CLR Managed Debugger (mdbg) Sample* [16].

Nad touto knihovnou byl naprogramován samostatný jednoduchý debugger. Vznik samostatného debuggeru místo jeho začlenění jako modulu do stávající aplikace je důsledek nutnosti definovat všem grafickým aplikacím, které využívají knihovnu *Mdbg,* atribut *MTAThread*. Tento atribut změní správu vláken celé aplikace, a proto bylo vhodné, aby byla tato část oddělena. Vznikla jednoduchá konzolová aplikace, jejíž chod je řízen pomocí příkazů zadávaných na standardní vstup a výstup je posílán na standardní výstup. Aplikace ovšem není určena pro samostatné fungování a reagování na příkazy uživatele, ale její fungování je řízeno nadřazenou vrstvou pomocí přesměrování standardního vstupu a výstupu. Proti samostatnému spuštění má zabudovanou ochranu a při takovém spuštění se sama ukončí. Aplikace má definovanou sadu příkazů, na které reaguje, pokud má příkaz nějaké parametry, jsou odděleny znakem „|“. Dostupných je 7 příkazů, jejichž tvary jsou:

- *d|path|name|sname|breakLine* – písmeno *d* značí příkaz začátku ladění, *path* je cesta k laděnému programu, *name* je jméno souboru programu s příponou, *sname* je jméno souboru se zdrojovým kódem také s příponou a *breakLine* je číslo řádku, na kterém začíná hlavní metoda programu a kam se umístí první *breakpoint*.
- *r|path|name|sname|breakLine* – písmeno *r* značí příkaz restartování ladění. Parametry jsou stejné jako v příkazu *d*
- *s* – příkaz posunutí o jeden řádek zdrojového kódu
- *sinto* – příkaz skoku do funkce
- *sout* – příkaz skoku ven z funkce
- *v|name* – příkaz zjištění hodnoty proměnné s názvem předaným v parametru *name*
- *q* – ukončení ladění

Nejvyšší vrstvou simulátoru je samotné okno simulátoru, spouštěné po kliku uživatele na ikonu simulace. Hlavní program simulátoru předá vygenerovaný zdrojový kód v jazyce C#, dále slovník obsahující pozice symbolů diagramu ve zdrojovém kódu, tento slovník je využíván pro označování pozice algoritmu v diagramu, třídu uchovávající data samotného projektu a posledním parametrem je cesta k adresáři, kde je zkompilovaný soubor programu.

Simulátor ještě před samotným spuštěním simulace musí provést několik skrytých operací. Dojde k uložení zdrojového kódu do souboru, což je nutné pro správnou funkci debuggeru, k upravení

zdrojového kódu spočívající v přidání čísla řádku na začátek každého řádku, vytvoření slovníku obsahujícího počáteční hodnoty všech použitých proměnných, dále nastavení grafického prostředí dle zvolené metody simulace.

Po nastavení všech nutných proměnných a inicializaci okna může uživatel zahájit simulaci stiskem tlačítka *start*. Po stisku simulátor provede několik operací, vytvoří a spustí nový proces aplikace nižší vrstvy – debuggeru. Odešle příkaz pro začátek ladění, na který by v ideálním případě měl debugger odpovědět číslem řádku, na kterém se aktuálně zastaví, v tomto případe to bude první řádek metody *main*. Po odeslání zmíněného příkazu je zavolána metoda *parseStatus()*. Tato metoda nastaví časovač, který určuje čas, který má debugger na odeslání odpovědi. Tento způsob čekání je zaveden z důvodu možnosti, že nastane situace, kdy debugger neodpoví hned. Tato situace může nastat, pokud probíhá složitý výpočet, program čeká na vstup od uživatele, nebo se vyskytla neočekávaná chyba, například zacyklení laděného algoritmu. Pokud debugger odpoví číslem řádku, dojde k označení zmíněného řádku v kódu, pomocí slovníku pozic symbolů v kódu dojde k označení aktuálně zpracovávaného symbolu a poté simulátor načte hodnoty proměnných z debuggeru a to příkazem *v*. Tento příkaz se zavolá nejdříve pro všechny parametry laděné funkce, poté pro všechny globální proměnné a nakonec pro všechny lokální proměnné. Po zahájení simulace má uživatel několik možností. Může využít funkci krokování, která odešle debuggeru příkaz *s* pro posunutí o jeden řádek v kódu a následně bude zavolána již popsaná metoda *parseStatus()*. Uživatel má možnost řídit krokování funkcemi *stepIn*, *stepOut*, pro skok do, nebo ven z funkce pokud je zavolána nad, nebo ve funkci. Další dostupnou funkcí je příkaz *stepSymbol*, který automaticky krokuje po řádcích zdrojového kódu, dokud nedojde na řádek s následujícím symbolem. Další užitečnou funkcí je automatické krokování v čase. Grafické rozhraní obsahuje posuvník, kterým si uživatel nastaví časový rozsah v rozmezí 300–3000 ms, a po vyvolání příkazů automatického krokování simulátor sám vyvolává příkaz step po uplynutí časového období.

4.7 Kompilace programů

Nevýhodou původní aplikace byla nutnost nastavit cesty k překladači jazyka C# a C (SDCC) ručně. Pro nezkušené uživatele mohly vzniknout problémy v případě, že cesta nebyla nastavena korektně. Proces kompilace aplikace pro obě platformy byl přepracován. Kompilátor SDCC má nově stálé místo v adresářové struktuře aplikace, konkrétně v adresáři SDCC, a proto již není nutné zadávat cestu a je volán automaticky z pevného umístění.

Překladač jazyka C# již není volán přímo pomocí svého souboru *scs.exe* ale programově, pomocí tříd ze jmenného prostoru *System.CodeDom.Compiler*. Kompilace probíhá v několika krocích, vytvoří se objekt typu *CodeDomProvider*, nastaví se parametry kompilátoru, jako je příznak, že se má vytvořit binární soubor s příponou *exe*, že se mají vytvořit ladící informace potřebné pro simulátor, či reference na použité knihovny atd. Následně se spustí kompilace, jejíž výsledek je uložen v objektu *CompilerResults*. Z tohoto objektu se také přečtou případné chyby vzniklé při kompilaci.

Kompilace aplikací s uživatelským rozhraním již není tak jednoduchá, jako kompilace aplikací konzolových. Je nutné vytvořit několik souborů popisujících projekt. Některé soubory jsou stejné pro všechny aplikace a jejich kódy jsou uloženy jako součást vývojového prostředí, některé jsou generované

- *App.xaml*, soubor, který může obsahovat zdroje nové aplikace, v tomto případě obsahuje jen prázdnou kostru.
- *App.xaml.cs*, definuje hlavní třídu aplikace, stejně jako v předchozím případě obsahuje pouze kostru.
- *WpfApp.csproj*, definuje vlastnosti projektu, například jména výstupních souborů, verzi framework, závislosti na dalších knihovnách a samotný postup kompilace.
- *MainWindow.xaml*, soubor s vygenerovaným kódem v jazyce XAML popisujícím uživatelské prostředí.
- *MainWindow.xaml.cs*, soubor s vygenerovaným kódem v jazyce C# se samotnou logikou aplikace.

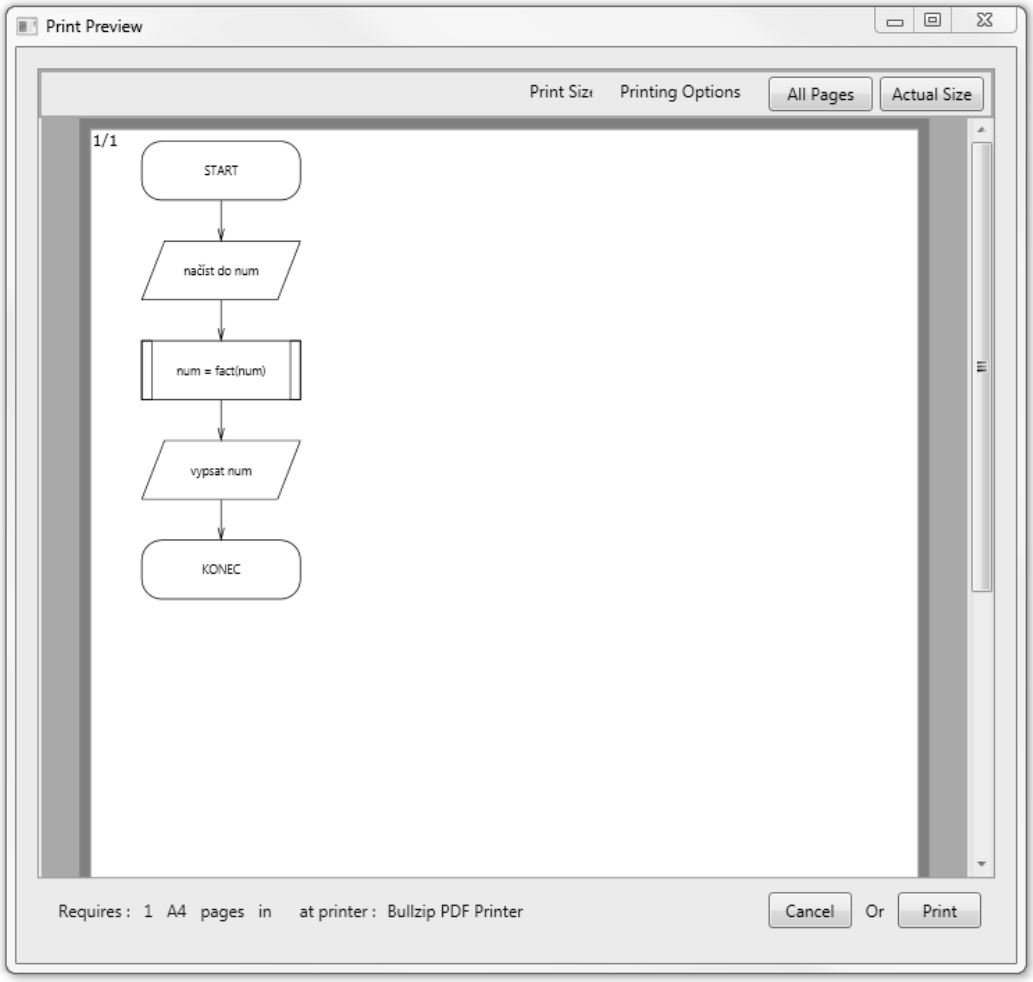
Samotný proces kompilace zajišťuje několik objektů, prvním z nich je *BuildRequestData*, ve kterém je uložena cesta k souboru *WpfApp.csproj*, verze kompilátoru, který se použije, a případně další parametry překladu v objektu *BuildParameters*. Důležitým parametrem je *Logger*, který zaznamenává události vzniklé v době překladu. Posledním důležitým objektem je *BuildResult*, ve kterém je uložen výsledek kompilace.

4.8 Tisk

S použitím nové technologie WPF se vázala také nutnost změnit způsob implementace tisku vývojových diagramů. Pro tento účel byla zvolena knihovna *WPF print engine* [17]. Jednou z funkcí této knihovny je umožnění tisku grafických komponent systému WPF. Její použití v aplikaci je jednoduché, protože tisknuta je celá oponenta *DiagramCanvas* pouze s několika úpravami. Je odstraněna mřížka a plocha je zmenšena na velikost odpovídající diagramu.

WPF print engine má vlastní rozhraní zobrazující náhled tisku s rozložením na jednotlivé stránky a několik dodatečných funkcí. Umožňuje dodatečnou změnu rozměrů tisknuté komponenty, takže diagram se dá zmenšovat či zvětšovat dle potřeby popřípadě množství papírů, umožňuje výběr orientace papíru na výšku, nebo na šířku a jeho rozměrů, popřípadě kvality tisků pomocí standardních dialogů Windows. Dále vypisuje jednoduchý přehled o zvolených nastaveních a počtu potřebných papírů.

Nevýhodou je složitá lokalizace prostředí *WPF Print engine*, ale prostředí je tak jednoduché a intuitivní, že lokalizace není nutná.

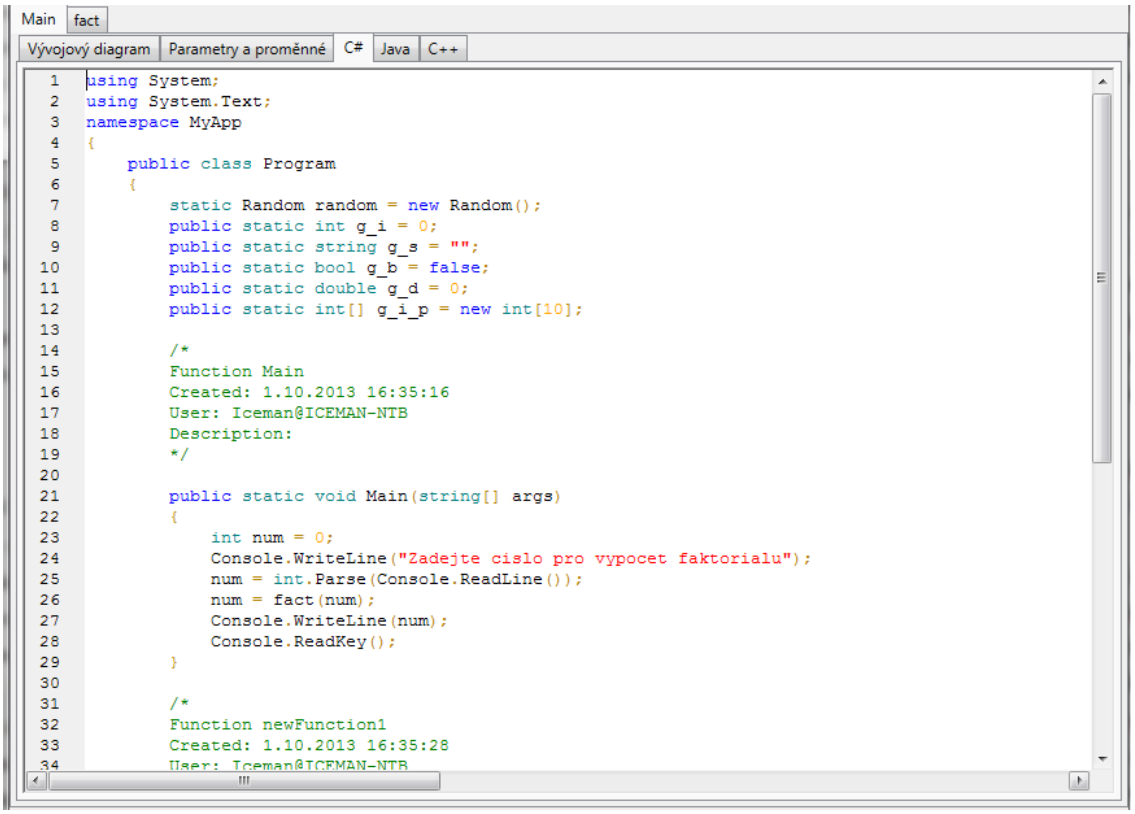


Obrázek 30: Prostředí WPF print engine

4.9 Zvýraznění syntaxe kódu

Další užitečná vlastnost, která v původní aplikaci chyběla, je zvýraznění syntaxe zobrazovaných vygenerovaných zdrojových kódů. Pro tento účel je také použita samostatná knihovna a to projekt Scintilla.NET [18]. Scintilla je grafická komponenta obsahující velmi pokročilé funkce pro tvorbu a editaci zdrojových kódů. Na jejím základu byl postaven i editor kódů v multiplatformním projektu pro vývoj .NET aplikací Mono. Mezi těmito funkcemi je například nápověda při psaní zdrojových kódů, automatické doplňování kódů, podpora schránky a příkazů *undo* a *redo*, zobrazování grafických *breakpoint*, popřípadě záložek, číslování řádků, export, tisk a mnoho dalších. Z této rozsáhlé škály funkcí jsou v aplikaci použity pouze dvě, protože komponenta slouží pouze pro zobrazení kódů a ne editaci. Je to zvýraznění syntaxe, kterou komponenta umí pro mnoho programovacích jazyků, a zobrazení čísel řádků.

Nevýhodou komponenty je to, že je napsána pro systém Windows Forms. Aby mohla být použita v novějším WPF, musí být umístěna uvnitř elementu *WindowsFormsHost.*



Obrázek 31: Komponenta Scintilla.NET

4.10 Podpora lokalizace prostředí

Existuje více přístupů pro lokalizaci aplikací nad knihovnou WPF. V této práci byl zvolen jeden z nejjednodušších a to pomocí lokalizovaných slovníků. Hlavní princip spočívá v tom, že všechny texty v aplikaci nejsou napsány přímo v kódu XAML, ale jsou načítány jako dynamické zdroje (*DynamicResource*) ze slovníku zdrojů. Výhodami tohoto přístupu je velice jednoduchá implementace bez nutnosti použití externích knihoven či nástrojů a okamžitá aktualizace všech textů po výběru jazyka bez nutnosti restartu aplikace. Existuje ale i několik nevýhod. Největší z nich je nutnost překompilování celého programu při přidávání jazyka či úpravách ve slovníku, to je zapříčiněno způsobem uložení souborů s lokalizacemi, ty jsou vloženy přímo do binárního souboru aplikace. Další možnou nevýhodou je nepřehlednost vytvořených slovníků s jazyky, pokud v aplikaci přibude nový text, který je nutné lokalizovat, je ho nutné přidat do všech již vytvořených slovníků. Samotný slovník s konkrétní lokalizací je samostatný soubor napsaný v jazyku XAML, obsahuje jeden kořenový element *ResourceDictionary*, který obsahuje pouze elementy *system:String*, což jsou konkrétní lokalizované texty aplikace označená pomocí atributu *Key*, pomocí kterého jsou přiřazeny grafickým komponentám aplikace. Takto napsané slovníky ještě musejí být zařazeny do zdrojů aplikace. Toho je docíleno upravením souboru *App.xaml* obsahující globální zdroje celé aplikace. V tomto souboru je element *Application.Resources* obsahující element *ResourceDiactionary*, do kterého budou vytvořené slovníky přidány. Toho je docíleno použitím elementu *ResourceDiactionary.MergedDictionaries*

obsahující reference na soubory s lokalizacemi. Princip zpracování referencovaných slovníků spočívá v tom, že jsou zpracovávány postupně, jak jsou pod sebou zapsány. Slovníky obsahují lokalizované texty se stejnými klíči, při načtení slovníku se zpracují texty, a pokud zdroje již obsahují text se stejným klíčem, tento text je přepsán. Aktivní zůstanou pouze texty zapsané ve slovníku referencovaném na posledním místě, protože jím jsou přepsány všechny poslední hodnoty.

Na tomto principu následně pracuje i aplikování lokalizace za běhu. Ze zmíněného hlavního slovníku aplikace je vyjmut slovník, který chce uživatel použít a následně je vložen na poslední místo, takže přepíše všechny původní texty. Tím, že jsou použity dynamické zdroje, všechny texty se poté automaticky aktualizují dle slovníku.

4.11 Ukládání projektů do souboru

Princip ukládání rozpracovaných projektů do souborů na disku zůstal zachován, ale technologie k tomuto účelu použitá byla přepracována. V původní aplikaci byl použit *BinaryFormatter*, který ukládá objekty v binárním formátu. Jeho nevýhodou ale je rozpadnutí referencí mezi serializovanými objekty. Jelikož se při ukládání projektu serializuje objekt *ProjectData*, který obsahuje seznamy objektů funkcí, proměnných, grafických komponent a dalších a jehož funkce je na referencích založena, reference musely být při následné deserializaci znovu navázány. Obnovení referencí nebyl triviální problém a vyžadovalo poměrně složitou vnitřní logiku. Nově je pro ukládání použit *DataContractSerializer* z jmenného prostoru *System.Runtime.Serialization.DataContractSerializer*. *DataContractSerializer* ukládá objekty do proudu ve formátu XML s tím, že zachovává reference mezi ukládanými objekty.

Třídy, které se budou pomocí tohoto serializátoru serializovat, se musejí označit atributem *DataContract*, který značí, že třídy obsahují datové proměnné určené pro uložení. Všechny tyto proměnné musejí být označeny také a to atributem *DataMember*. Mimo těchto dvou atributů se při použití *DataContractSerializer* vyskytuje ještě atribut třetí a to *KnownType*. Tento atribut se používá pro informování serializátoru o tom, jaké datové typy jsou v hlavním objektu referencovány a budou serializovány společně s ním. V případě konkrétně této aplikace jsou tyto atributy přidány k třídě *ProjectData* a mají tvar například *KnownTypeAttribute(typeof(SymbolData))*.

Jak již bylo zmíněno, *DataContractSerializer* ukládá objekty do proudu ve formátu XML. Výhodou tedy je, že výsledný soubor je čitelný i pro uživatele, malou nevýhodou je nárůst paměťové náročnosti, uložené projekty mají velikost řádově ve stovkách kB. Tato nevýhoda by se dala potlačit použitím komprimace výsledného proudu s tím, že by soubory přišly o svou čitelnost.

Aplikace umožňuje také ukládání funkcí do knihovny, které funguje na zcela stejném principu. I objektu *FunctionData*, který uchovává data funkce, jsou přidány atributy *KnownType*, takže je možné ho uložit samostatně. Soubory funkcí v knihovně obsahují pouze tento serializovaný objekt.

4.12 Indikace zaneprázdnění aplikace

V aplikaci se vyskytují úlohy, které jsou časově náročné, a bylo vhodné vyřešit, jak dát uživateli najevo, že aplikace pracuje a nepřestala odpovídat. Pro tento účel byla zvolena komponenta *BusyIndicator* z knihovny *Xceed Wpf Toolkit* [19]. Tato knihovna obsahuje velmi pokročilé grafické komponenty, jako je komponenta pro interaktivní výběr barvy *ColorPicker*, komponenta pro zadávání čísel pomocí tlačítek nahoru a dolů *NumericUpDown*, komponenta pro ošetřený vstup dle masky *MaskedTextBox*, nebo i jednoduchou kalkulačku.

Mimo jednoduchého vyvolání *BusyIndicatoru*, který se zobrazí přes celé prostředí a uprostřed má grafický pruh indikující, že aplikace pracuje, bylo nutné časově náročné úlohy přesunout do samostatných vláken, aby bylo zajištěno, že aplikace bude odpovídat a prostředí se nepřestane reagovat. Pro tento účel byla vytvořena statická metoda se čtyřmi parametry, prvním z nich je reference na *BusyIndicator* jež se vyvolá, druhým je zpráva k zobrazení, třetím je objekt typu *Action*, který uchovává metodu vyvolanou na pozadí v jiném vlákně a čtvrtým je *Action* s metodou vyvolanou po ukončení práce. Samotná metoda je vlastně jen kostra, která používá komponentu *BackgroundWorker*, které předává popsané funkce. *BackgroundWorker* je komponenta, která umožňuje vyvolat nějakou akci asynchronně v novém vlákně bez ručního definování vlákna. Programátor se tak při jednoduchých úlohách nemusí zatěžovat s tvorbou vláken [20].

Popsaný přístup je použit například při kompilaci zdrojových kódů, spouštění vygenerovaných aplikací popřípadě načítání souborů. Vyskytla se ale jedna oblast, kde zmíněný postup nejde použít.

Grafické rozhraní aplikace je z větší části uspořádáno do komponenty se záložkami, kde každá záložka obsahuje data jedné funkce. Tak jak je možné vytvářet funkce dynamicky, je nutné také příslušné grafické rozhraní funkce v záložce generovat automaticky za běhu. Toto grafické rozhraní je ale poměrně složité a jeho vytvoření za běhu zabere nějaký čas, kdy aplikace přestane odpovídat. V tomto čase není možné ani použít *BusyIndicator*, protože se sice zobrazí, ale přestane reagovat stejně jako zbytek grafiky. Použití vláken tento problém neřeší, protože grafické rozhraní aplikace může být tvořeno pouze v hlavním vlákně. Tento neduh se projeví hlavně při otevírání složitého projektu obsahujícího mnoho metod, protože aplikace musí vygenerovat mnoho dynamické grafiky. Snahou bylo proces otevírání souborů projektů co nejvíce optimalizovat, protože bez optimalizace při otevírání složitějšího projektu aplikace přestane odpovídat někdy i na dvě sekundy či déle, což je příliš dlouho. Optimalizovány jsou různé části procesu. Samotné načtení souboru z disku lze provést v samostatném vlákně a zobrazit *BusyIndicator*. Poté ale musí být všem funkcím vytvořena grafika. Optimalizace této části spočívá v zjištění, zda rozhraní nějaké funkce již nebylo vygenerováno a lze ho zachovat. Generování zbývajících rozhraní je přidáno do fronty *Dispatcheru* aplikace. Použitím *Dispatcheru* je zajištěno, že generovaná rozhraní se budou zobrazovat postupně tak, jak jsou tvořena, a aplikace zčásti reaguje na pokyny od uživatele. S tím jak se postupně zobrazují záložky s vygenerovanými rozhraními funkcí, uživatel vidí, že je projekt zpracováván a aplikace pracuje.

4.13 Validace vstupů od uživatele

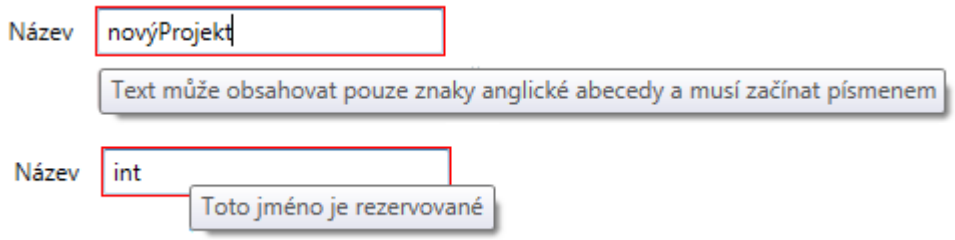
V kapitole věnované datové vazbě bylo zmíněno, že vazba může být obohacena validací hodnot. Validace je implementována i u většiny vstupů od uživatele v aplikaci. Síla validace nespočívá jen v navrácení původní hodnoty do komponenty, kam byla zadána nevalidní hodnota, ale umožňuje také v případě nevalidního vstupu aplikovat šablonu komponenty, která změní její vzhled tak, aby uživatel měl grafickou odezvu na špatný vstup. Mimo změny grafiky je také možné zpracovat zprávu, která je automaticky předána v případě nevalidního vstupu a její obsah nastavit třeba jako kontextovou nápovědu komponenty (*ToolTip*). Jádrem validace je validační pravidlo, což je třída odvozená od třídy *ValidationRule*. Odvozená třída musí předefinovat metodu *Validate*, jejíž návratovou hodnotou je objekt typu *ValidationResult* s výsledkem, zda je vstup validní a případnou zprávou pro uživatele. Vytvořené validační pravidlo se následně použije v XAML kódu požadované komponenty. Výhodou je možnost komponentě definovat více validačních pravidel, která se aplikují sekvenčně.

Validační pravidla jsou v aplikaci použita na mnoha místech a to například při zadávání přímých hodnot ve vlastnostech symbolů diagramu, při zadávání jmen projektů, nebo funkcí, či pro jednoduchou kontrolu, zda je vyplněno nějaké pole, které nesmí zůstat prázdné.

Příkladem může být vyplnění jména projektu. To nesmí zůstat prázdné, takže je aplikováno vytvořené pravidlo *EmptyStringValidation*, dále musí obsahovat pouze znaky anglické abecedy, to je kontrolováno pravidlem *AlphabetCharactersStringValidation* a jako poslední je kontrola, zda zvolené jméno není rezervovaným slovem dle pravidla *ReservedNameValidation.*

```
<TextBox x:Name="ProjectName">
   <TextBox.Text>
      <Binding Path="Name" UpdateSourceTrigger="PropertyChanged" >
         <Binding.ValidationRules>
            <validation:EmptyStringValidation  ValidatesOnTargetUpdated="True" />
            <validation:AlphabetCharactersStringValidation
                        ValidatesOnTargetUpdated="True" />
            <validation:ReservedNameValidation ValidatesOnTargetUpdated="True" />
         <Binding.ValidationRules>
      </Binding>
   </TextBox.Text>
</TextBox>
```

Obrázek 32: Definice validačních pravidel



Obrázek 33: Informace o nevalidním vstupu

4.14 Systém nápovědy

Možným nedostatkem systému WPF je absence integrovaného systému kontextové nápovědy. Programátor má možnost implementovat nápovědu pomocí *ToolTip*, která je implementována i v této aplikaci. Některé prvky aplikace ale bylo vhodné obohatit komplexnější kontextovou nápovědou provázanou s manuálem.

Tento problém byl vyřešen pomocí identifikátorů, které byly přiřazeny komponentám a při vyvolání manuálu aplikace směřovaly uživatele na konkrétní místo v dokumentu. Toho bylo docíleno pomocí statické třídy, která definovala tzv. *AttachedProperty* klíčové slovo. *AttachedProperty* je vlastnost, která není definována přímo v objektu, ale může mu být přiřazena. Všem komponentám, umožňujícím směrování nápovědy bylo přiřazeno klíčové slovo, ve tvaru *help.klic*, které je při vyvolání nápovědy přeloženo na nadpis odstavce v nápovědě, na který bude uživatel směrován.

```
<Button  Command="New" local:Help.Keyword="help.new" />
```

Obrázek 34: Připojení klíčového slova ke komponentě

Samotný soubor nápovědy je vytvořen v aplikaci Microsoft Office Word a následně je přeložen do standardního souboru kompilované nápovědy Windows s příponou *chm*. Soubor nápovědy se vyvolá pomocí objekt *Help* ze jmenného prostoru *System.Windows.Forms*.

4.15 Použití vložených zdrojů

Embedded Resources, neboli vložené zdroje jsou soubory, které se zkompilují spolu se zdrojovým kódem aplikace a přibalí se do vzniklého spustitelného souboru. Výhodou použití vložených zdrojů je, že soubory, potřebné pro fungování aplikace, nejsou viditelné v adresářové struktuře, a tudíž nemůžou vzniknout problémy při jejich smazání. Tento přístup je využit ve třech případech.

Prvním z nich jsou soubory potřebné pro kompilaci projektů. U projektů pro mikroprocesor to je soubor s obslužnými rutinami periferií *pripravek.inc* a hlavičkový soubor definující konstanty mikroprocesoru *AT89C51CC03.h*. U projektů s grafickým uživatelským rozhraním to jsou soubory *App.xaml*, *App.xaml.cs*, *WpfApp.csproj* nutné k vytvoření projektu ke zkompilování.

Druhým případem jsou vložené knihovny. Většina knihoven, které aplikace využívá, je také přibalena jako *Embedded Resources*. Důvodem je jejich větší množství, které dělá kořenovou složku aplikace lehce nepřehlednou, a dále ochrana proti smazání. Načítání knihoven pro potřeby aplikace pak potřebuje speciální postup. Je využívána třída *EmbeddedAssemly* [21]. Při startu aplikace se musí všechny přiložené knihovny nahrát do paměti pomocí příkazu *EmbeddedAssembly.Load(cesta_ke_knihovne,nazev_souboru)* a poté musí být přidán uživatelský *AssemblyResolver*, který danou knihovnu zpřístupní, pokud ji aplikace vyžádá. Tento resolver pouze vyžádá knihovnu od třídy *EmbeddedAsesembly*.

Třetím případem je vložený soubor, který obsahuje rezervovaná jména, která nelze použít jako jména proměnných při tvorbě aplikací. Tento soubor má klasický textový formát, kde na jednom řádku je vždy jedno rezervované slovo, každý řádek je ukončen středníkem.

5 Závěr

V teoretické části této práce je popsána technologie Windows Presentation Foundation použitá v aplikaci, která s touto prací vznikla. Praktická část obsahuje popis implementace aplikace, která vznikla rozšířením aplikace vytvořené v rámci předchozí práce o prvky, které obohacují její funkčnost. Seznam hlavních implementovaných prvků je obsažen v prvních dvou bodech zadání a jsou jimi mimo jiné rozšíření cílových platforem o možnost tvorby aplikací s grafickým rozhraním, možnost práce s poli a vytvoření simulátoru navrhnutých algoritmů. Implementace každého z nich je popsána v kapitole obsahující implementační detaily.

V rámci posledního bodu zadání byla vytvořena sada ukázkových úloh, které jsou k práci přiloženy a jsou jimi různé matematické algoritmy, jako je výpočet faktoriálu, kvadratické rovnice a jiných, dále ukázky řadících algoritmů, ukázky jednoduchých grafických aplikací, jako je kalkulačka, nebo grafický program pro řešení již zmíněné kvadratické rovnice, ukázky algoritmů ovládajících různé periferie vývojové desky a také řešení několika úloh z projektu Euler, což je seznam mnoha matematických úloh dostupný ze stránky *http://projecteuler.net*.

Výsledkem práce je vývojové prostředí, které umožňuje tvorbu funkčních programů, jak pro platformu Windows, tak pro vývojovou desku s mikroprocesorem a to za použití pouze vývojových diagramů, bez nutnosti napsání jediného řádku zdrojového kódu. Aplikace je použitelná buď pro výuku programování jako učební pomůcka, nebo jako nástroj pro objevování principu programování pro laiky.

Použitá literatura

[1] HORÁK, Tomáš. *Programování pomocí grafických symbolů*. Liberec, 2012. Bakalářská práce. Technická univerzita v Liberci. Vedoucí práce Ing. Tomáš Martinec, Ph.D.

[2] ČSN ISO 5807. *Zpracování informací*: *Dokumentační symboly a konvence pro vývojové diagramy, diagramy toku dat, programu a systému, síťové diagramy systému a diagramy zdrojů systému*. První vydání. Praha: Český normalizační institut, 1996.

[3] HORÁK, Tomáš. *Programování pomocí grafických symbolů*. Liberec, 2013. Dostupné z: Přiložené CD. Diplomový projekt. Technická univerzita v Liberci. Vedoucí práce Ing. Tomáš Martinec, Ph.D.

[4] MICROSOFT. Introduction to WPF. *Microsoft developer network* [online]. 2014 [cit. 2014-01-10]. Dostupné z: msdn.microsoft.com/en-us/library/aa970268(v=vs.100).aspx.

[5] PETZOLD, Charles. *Mistrovství ve Windows Presentation Foundation*. Vyd. 1. Brno: Computer Press, 2008, 928 s. ISBN 978-80-251-2141-2.

[6] MICROSOFT. Application class. *Microsoft developer network* [online]. 2014 [cit. 2014-01-12]. Dostupné z: msdn.microsoft.com/en-us/library/system.windows.application(v=vs.110).aspx.

[7] HERCEG, Tomáš a Tomáš JECHA. Architektura WPF - Dispatcher. *DetNETportál* [online]. 26. 1. 2012 [cit. 2014-01-20]. Dostupné z: www.dotnetportal.cz/clanek/197/Architektura-WPF-Dispatcher.

[8] HERCEG, Tomáš a Tomáš JECHA. Architektura a objektový model WPF. *DetNETportál* [online]. 9. 2. 2012 [cit. 2014-01-20]. Dostupné z: http://www.dotnetportal.cz/clanek/199/Architektura-a-objektovy-model-WPF.

[9] HERCEG, Tomáš a Tomáš JECHA. Jazyk XAML. *DetNETportál* [online]. 2. 2. 2012 [cit. 2014-01-20]. Dostupné z: http://www.dotnetportal.cz/clanek/198/Jazyk-XAML.

[10] MICROSOFT. XAML Overview (WPF). *Microsoft developer network* [online]. 2014 [cit. 2014-01-15]. Dostupné z: msdn.microsoft.com/en-us/library/ms752059(v=vs.100).aspx.

[11] HERCEG, Tomáš a Tomáš JECHA. Základy pozicování. *DetNETportál* [online]. 23. 2. 2012 [cit. 2014-01-23]. Dostupné z: http://www.dotnetportal.cz/clanek/205/Zaklady-pozicovani.

[12] JIRAVA, Jarda. WPF Layout engine. *Xaml.cz* [online]. 19. 01. 2010 [cit. 2014-02-10]. Dostupné z: http://xaml.cz/wpf/wpf-layout-engine.

[13] MICROSOFT. Commanding Overview. *Microsoft developer network* [online]. 2014 [cit. 2014-02-20]. Dostupné z: msdn.microsoft.com/en-us/library/ms752308(v=vs.110).aspx.

[14] MICROSOFT. ApplicationCommands Class. MICROSOFT. *Microsoft developer network* [online]. 2014 [cit. 2014-02-20]. Dostupné z: http://msdn.microsoft.com/en-us/library/system.windows.input.applicationcommands(v=vs.110).aspx.

[15] ŠTURALA, Aleš. WPF - vytváříme kontroly. BLÁHA, Michal. *Vyvojar.cz: Vývojáři sobě...* [online]. 27. 2. 2007 [cit. 2014-02-21]. Dostupné z: http://www.vyvojar.cz/Articles/460-6-wpf-vytvarime-kontroly.aspx.

[16] MICROSOFT. CLR Managed Debugger (mdbg) Sample 4.0. *Microsoft download center* [online]. 31. 1. 2011 [cit. 2014-03-02]. Dostupné z: http://www.microsoft.com/en-us/download/details.aspx?id=2282.

[17] TALUKDER, Saraf. WPF Print Engine: Part I. CODEPROJECT. *Code Project: For those who code* [online]. 21. 8. 2011 [cit. 2014-03-10]. Dostupné z: http://www.codeproject.com/Articles/238135/WPF-Print-Engine-Part-I.

[18] LUSSER, Jacobs. ScintillaNET. *CodePlex: Project hosting for open source software* [online]. 17. 2. 2014 [cit. 2014-03-13]. Dostupné z: http://scintillanet.codeplex.com.

[19] XCEED. Extended WPF Toolkit™ Community Edition. MICROSOFT. *CodePlex: Project hosting for open source software* [online]. 20. 2. 2014 [cit. 2014-03-13]. Dostupné z: https://wpftoolkit.codeplex.com.

[20] BAYER, Jürgen. *C# 2005 - Velká kniha řešení*. Brno: Computer Press, 2007, ISBN 978-80-251-1620-3.

[21] DRIANCS. Load DLL From Embedded Resource. CODEPROJECT. *Code Project: For those who code* [online]. 17. 1. 2013 [cit. 2014-03-13]. Dostupné z: http://www.codeproject.com/Articles/528178/Load-DLL-From-Embedded-Resource.

Příloha A: CD

Obsah CD:

- Text práce
- Aplikace (pro správnou funkčnost je nutné aplikaci před spuštěním překopírovat na pevný disk)
- Zdrojový kód
- Ukázky projektů, obsaženy také v adresáři s aplikací, podadresáři *projects*
- Text diplomového projektu

Příloha B: Ukázkové úlohy

K programu byla vytvořena sada ukázkových úloh, které se nacházejí na CD v samostatném adresáři a také v adresáři aplikace v podadresáři *projects*.

- Aplikace s grafickým rozhraním
  - Řešení kvadratické rovnice
  - Práce s ukazatelem průběhu
  - Práce s přepínacími tlačítky
- Matematické algoritmy
  - Kvadratická rovnice
  - Faktoriál
- Vyhledávací algoritmy
- Řešení úlohy 1–9 z projektu Euler
- Ovládání mikroprocesoru
  - Zobrazení binárního čísla na *LedBaru*
  - Ukázky práce s diodami
  - Generování náhodných čísel
  - Ukázka použití sériové linky
  - Ovládání žárovky pomocí pulzně šířkové modulace

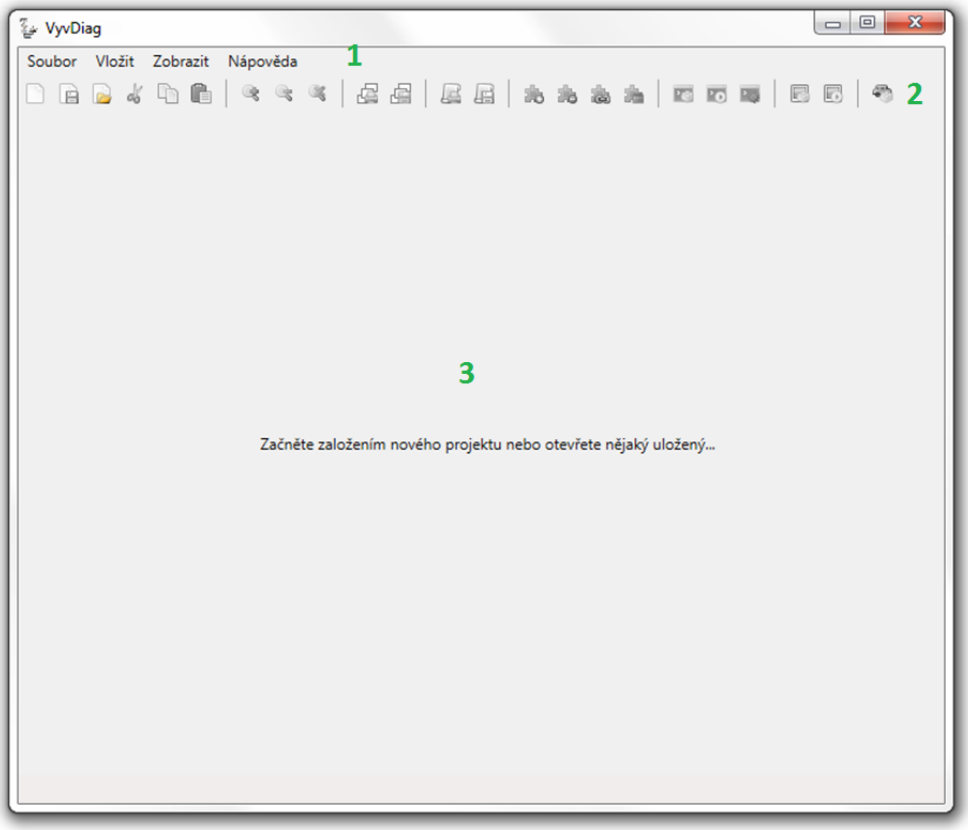
Mimo ukázkových úloh bylo vytvořeno také několik samostatných ukázkových funkcí, které je možné do projektu importovat pomocí knihovny. Nacházejí se na CD v adresáři aplikace v podadresáři *library*.

- Řadící algoritmy
  - Bubble sort
  - Select sort
  - Insert sort
  - Quick sort
- Mocnina čísla
- Faktoriál
- Řešení kvadratické rovnice
- Vytisknutí pole na konzoli
- Naplnění pole náhodnými čísly
- Zjištění, zda vstup je palindrom
- Zjištění, zda vstup je prvočíslo

Příloha C: Vývojové diagramy – Manuál

Aplikace Vývojové diagramy umožňuje tvorbu jednoduchých aplikací tří typů pomocí grafického jazyka – vývojových diagramů. Aplikace vznikla jako diplomová práce na Technické univerzitě v Liberci, autorem je Tomáš Horák a vedoucím práce je Ing. Tomáš Martinec, Ph.D.

1 Prostředí



Hlavní okno aplikace je rozděleno na tři části. 1. hlavní menu, 2. nástrojová lišta, 3. prostor pro tvorbu projektu.

1.1 Hlavní menu

Struktura hlavního menu obsahuje většinu funkcí, které aplikace poskytuje

- Soubor
  - Nový
  - Otevřít
  - Uložit
  - Uložit jako
  - Tisk
  - Zavřít projekt
  - Konec

- Vložit
    - Funkci
    - Symbol
        - Start
        - Konec
        - Podmínka
        - Cyklus
        - Funkce
        - Vstup/Výstup
        - Zpracování
    - Grafickou komponentu
        - Popisek
        - Tlačítko
        - Zaškrtávací tlačítko
        - Výběrové tlačítko
        - Ukazatel průběhu
        - Vstup textu
- Zobrazit
    - Vlastnosti projektu
    - Knihovnu funkcí
    - Výběr jazyka rozhraní
- Nápověda
    - Zobrazit nápovědu
    - Klávesové zkratky
    - O programu

1.2 Nástrojová lišta

Nástrojová lišta obsahuje zkratky k nejpoužívanějším funkcím a je rozdělena na části pomocí oddělovačů.

1.2.1 Základní funkce

- Vytvoření nového projektu
-  Uložení aktuálního projektu
-  Otevření nového projektu
-  Vyjmout do schránky
-  Kopírovat do schránky
-  Vložit ze schránky

1.2.2 Úprava zobrazení

Funkce slouží k zvětšení či zmenšení vývojových diagramů nebo zdrojových kódů

-  Přiblížit
-  Oddálit
-  Návrat na původní hodnotu přiblížení

1.2.3 Funkce pro práci s diagramem

Tyto funkce jsou aktivní, pouze pokud je zobrazen vývojový diagram

-  Tisk diagramu
-  Uložení diagramu do souboru

1.2.4 Funkce pro práci se zdrojovým kódem

Tyto funkce jsou aktivní, pouze pokud je zobrazen zdrojový kód

-  Tisk kódu
-  Uložení kódu do souboru

1.2.5 Funkce pro práci s funkcemi

-  Vytvoření nové funkce
-  Smazání aktuálně otevřené funkce
-  Kontrola validity aktuálně otevřené funkce
-  Export aktuálně otevřené funkce do knihovny

1.2.6 Funkce pro práci s konzolovým projektem

Tyto funkce jsou aktivní, pouze pokud je otevřen konzolový projekt

-  Kompilace projektu a vytvoření spustitelného souboru
-  Kompilace projektu, vytvoření spustitelného souboru a jeho spuštění
-  Kompilace projektu, vytvoření spustitelného souboru a zahájení simulace

1.2.7 Funkce pro práci s grafickým projektem

Tyto funkce jsou aktivní, pouze pokud je otevřen grafický projekt

-  Kompilace projektu a vytvoření spustitelného souboru
-  Kompilace projektu, vytvoření spustitelného souboru a jeho spuštění

1.2.8 Funkce pro práci s projektem pro mikroprocesor

Tato funkce je aktivní, pouze pokud je otevřen projekt pro mikroprocesor

-  Kompilace projektu a vytvoření binárního souboru

2 Typy projektů

Aplikace umožňuje vytvářet projekty pro tři platformy, konzolové aplikace nad .NET Framework, aplikace s grafickým rozhraním nad stejným Framework s knihovnou Windows Presentation Foundation a aplikace pro vývojovou desku s mikroprocesorem Atmel AT89C51CC03.

2.1 Konzolové aplikace

Konzolová aplikace je aplikace bez grafického rozhraní ovládaná pouze textově z konzole. Konzole tak představuje vstup i výstup vytvořené aplikace. Aplikace má pro konzolové projekty nejvíce funkcí. Umožňuje generování zdrojových kódu v jazycích C#, Java a C++, umožňuje jejich kompilaci a spouštění a jednoduchou simulaci.

2.2 Aplikace s grafickým rozhraním

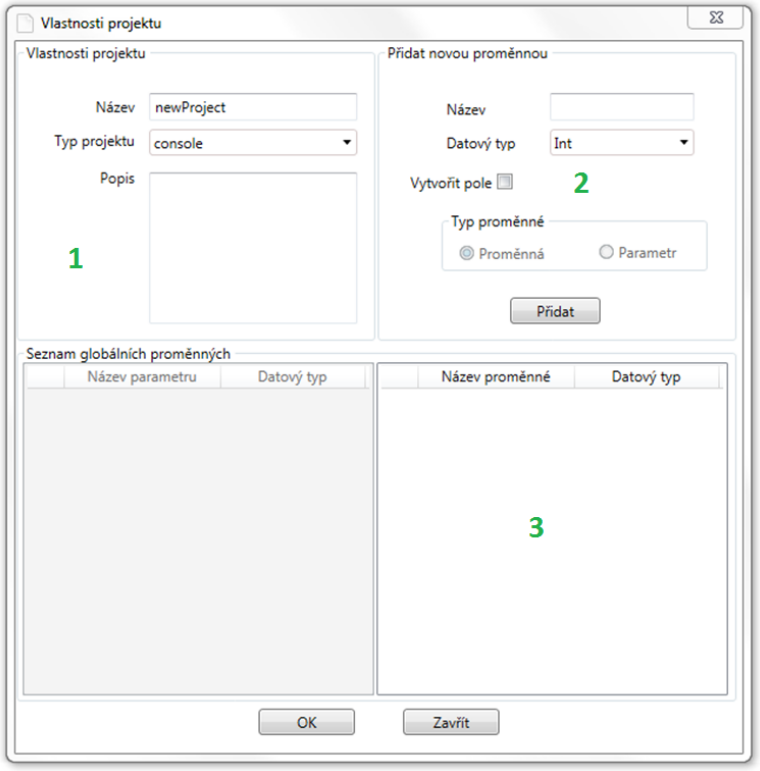
Aplikace s grafickým rozhraním je klasická desktopová aplikace. Vstupy a výstupy jsou zprostředkovány grafickými komponentami. Aplikace umožňuje navrhnout grafické rozhraní, generovat zdrojové kódy v jazyce C# a XAML a vytvořené projekty zkompilovat a spustit.

2.3 Aplikace pro vývojovou desku

Aplikace pro vývojovou desku nejsou klasické spustitelné soubory, ale binární soubory vhodné pro nahrání do zmíněné desky, případně jejího simulátoru. Vstupy a výstupy jsou hardwarové komponenty, ovládané speciálními proměnnými. Aplikace umožňuje vygenerovat zdrojové kódy v jazyce C a vytvořit binární soubor.

3 Vytvoření nového projektu

Vytvoření nového projektu může být vyvoláno z hlavního menu, položkou Soubor – Nový projekt, nebo ikonou bíle stránky z nástrojové lišty, případně klávesovou zkratkou *Control+N*. Tato akce vyvolá okno vlastností projektu.



Okno je rozděleno na tři části, 1. základní vlastnosti projektu, 2. tvorba globálních proměnných, 3. seznam globálních proměnných.

3.1 Základní vlastnosti projektu

Mezi základní vlastnosti projektu patří název, pro který platí speciální pravidla pro tvorbu názvů, typ projektu, který určuje platformu, pro kterou je aplikace tvořena, a popis projektu.

3.2 Tvorba globálních proměnných

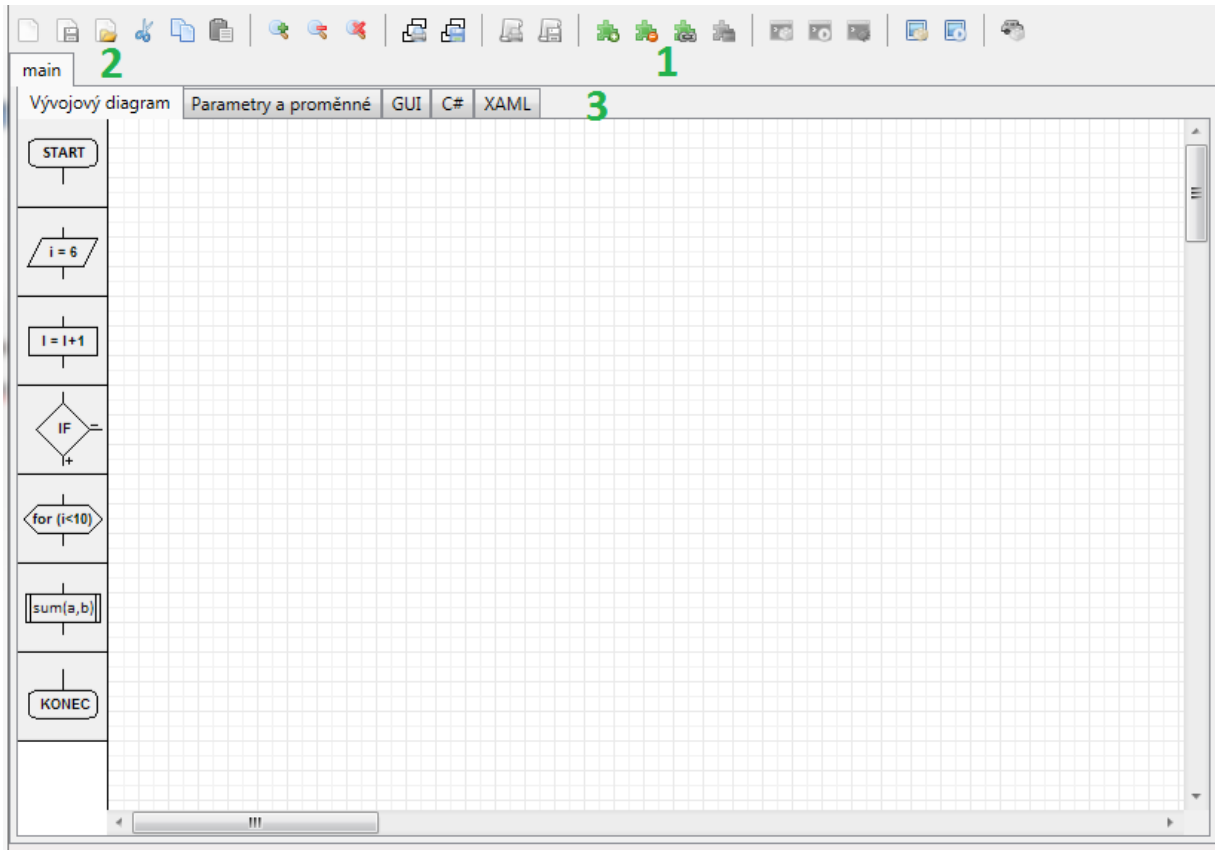
Při zakládání projektu je možné definovat globální proměnné. Takové proměnné budou dostupné ze všech funkcí programu. Pro tvorbu globálních proměnných platí stejná pravidla, jako pro tvorbu lokálních proměnných.

3.3 Seznam globálních proměnných

Seznam globálních proměnných obsahuje přehled již vytvořených globálních proměnných.

4 Práce s funkcemi

Po založení nového projektu je automaticky vytvořena hlavní funkce – *Main*, se kterou se pracuje úplně stejně jako se všemi ostatními funkcemi, aplikace pro ni vytvoří záložku v hlavním okně.



Pro každou funkci se v grafickém rozhraní aktivují některé ovládací prvky. 1. ikony se zeleným dílkem puzzle, 2. záložky vytvořených funkcí, 3. záložky s nástroji pro správu funkcí.

4.1 Ikony se zeleným dílkem puzzle

Na hlavním panelu jsou čtyři ikony pro práci s funkcemi. První, se znakem plus slouží k přidání nové funkce. Druhá se znakem mínus, slouží k odebrání aktuálně označené funkce a je aktivní pouze pro funkce jiné než *Main*. Třetí se znakem řetězu, slouží k otestování validity diagramu funkce. Čtvrtá se znakem krabičky slouží k exportování funkce do knihovny funkcí a je aktivní pouze pro funkce jiné než *Main*.

4.2 Záložky vytvořených funkcí

Horní řada záložek obsahuje jednu záložku pro každou vytvořenou funkci

4.3 Záložky s nástroji pro správu funkcí

Dolní řada záložek obsahuje nástroje pro správu funkcí, které se můžou lišit podle tvořeného projektu:

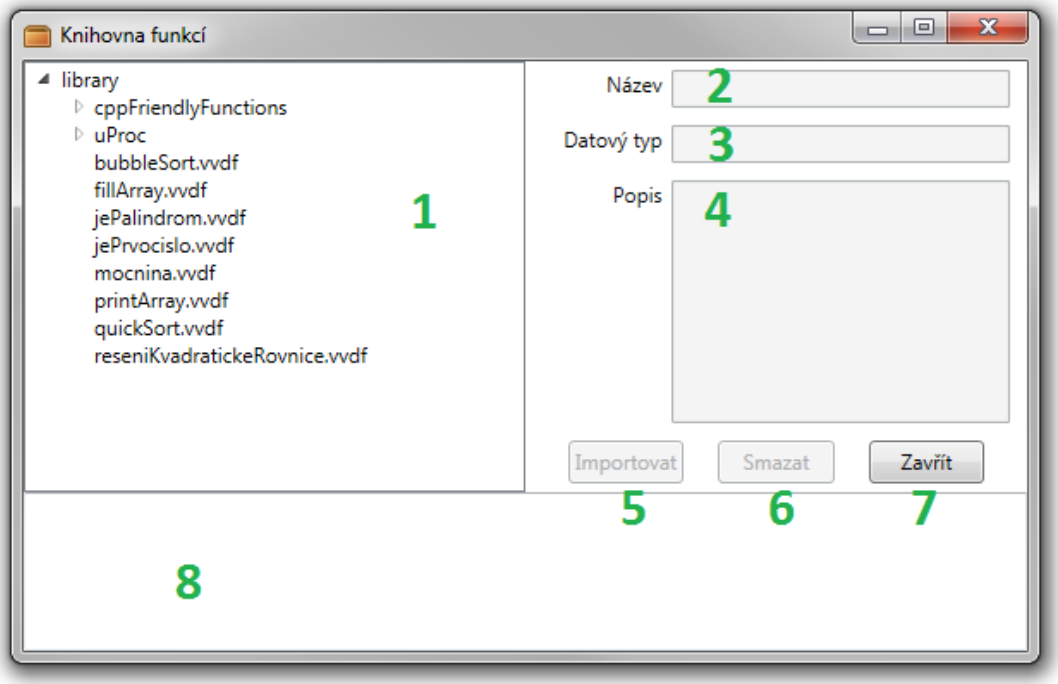
- Vývojový diagram
- Parametry a proměnné
- GUI
- C#
- C++

- Java
- C
- XAML

4.4 Knihovna funkcí

Aplikace umožňuje vytvořené funkce (jiné než *Main*) exportovat do tzv. knihovny funkcí. Tato funkce je přístupná po kliku na ikonu zeleného dílku puzzle s obrázkem krabice. Funkce je exportována do souboru s příponou *vvdf* a aby bylo možné ji zobrazit v knihovně, musí být uložena do složky *library* v kořenové složce aplikace.

Funkce již dříve exportované do knihovny je možné do projektu importovat. Rozhraní knihovny se vyvolá z hlavního menu položkou Zobrazit – knihovna funkcí.



Rozhraní knihovny funkcí je rozděleno na několik částí, 1. dostupné funkce, 2.,3.,4. informace o vybrané funkci, 5.,6.,7. ovládací tlačítka, 8. informační pole.

4.4.1 Dostupné funkce

Pole obsahuje stromovou strukturu funkcí tak, jak je uložena ve složce *library* umístěné v kořenové složce aplikace. Funkce k importu je vybrána jednoduchým klikem na její název.

4.4.2 Informace o vybrané funkci

Po vybrání funkce jsou zobrazeny informace, jako je její název, datový typ návratové hodnoty a popis.

4.4.3 Ovládací tlačítka

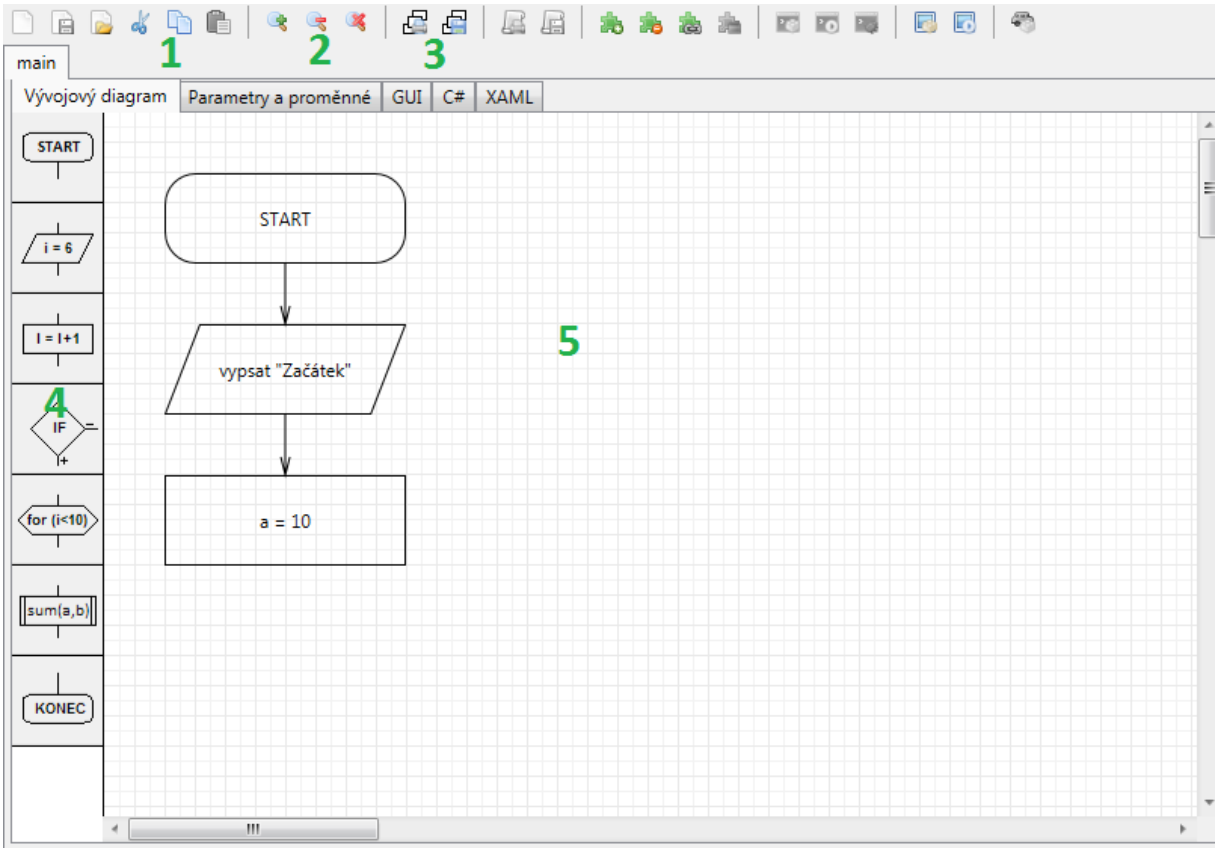
Tři dostupná ovládací tlačítka slouží k importu funkce do projektu smazání funkce z knihovny a zavření rozhraní knihovny.

4.4.4 Informační pole

Informační pole zobrazuje podrobnosti importu, jako jsou přiložené funkce, globální proměnné popřípadě kolize.

5 Tvorba vývojového diagramu

Logika každé funkce je definována vývojovým diagramem, který se nachází v první záložce nástrojů pro správu funkcí.



Při zobrazení pracovní plochy pro tvorbu diagramu je aktivováno několik nástrojů z nástrojové lišty, 1. schránka dostupná pomocí ikony kopírovat, vyjmou a vložit a dále známými klávesovými zkratkami, 2. možnost přibližování a oddalování pracovní plochy, 3. funkce pro tisk diagramu, nebo jeho export jako obrázku. Samotná plocha pro tvorbu diagramu je rozdělena na dvě části, 4. seznam dostupných symbolů, 5. plocha pro diagram.

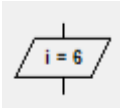
5.1 Seznam dostupných symbolů

Vývojový diagram se vytváří přetahováním symbolů z jejich seznamu na pracovní plochu. Dostupných je 7 symbolů, každý s unikátní funkcí a grafikou.

5.1.1 Symbol start

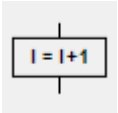
Tento symbol slouží k definování začátku každého algoritmu a je povinný.

5.1.2 Symbol pro vstup a výstup dat



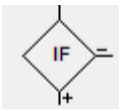
Tento symbol slouží k definování operací pro načítání dat od uživatele, výpisu dat programu a také definování návratové hodnoty funkce.

5.1.3 Symbol zpracování



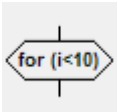
Tento symbol slouží k definování operací nad daty.

5.1.4 Symbol podmínky



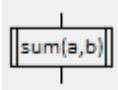
Tento symbol slouží k větvení programu pomocí podmínky.

5.1.5 Symbol cyklu



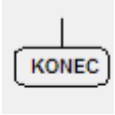
Tento symbol slouží k definování opakujících se akcí – cyklů. Dostupné jsou cykly s pevným počtem opakování a s podmínkou.

5.1.6 Symbol funkce



Tento symbol slouží k definování použití vytvořené funkce v algoritmu.

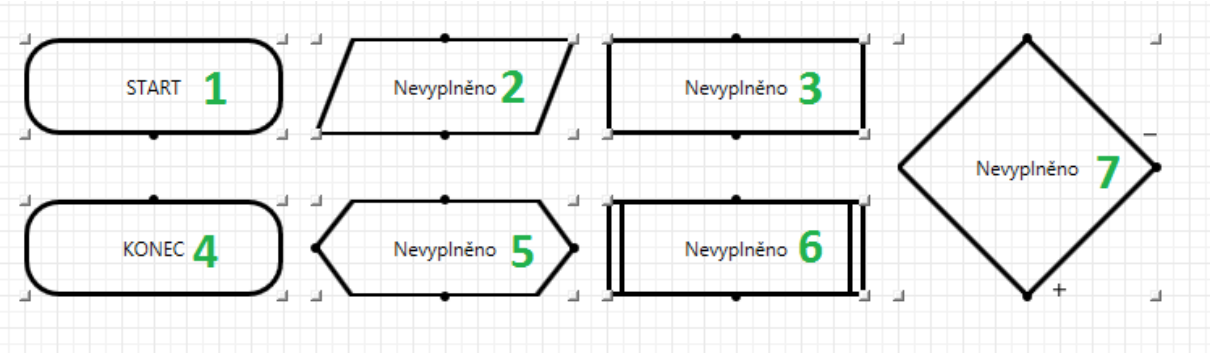
5.1.7 Symbol konec



Tento symbol definuje konec každého algoritmu a je povinný.

5.2 Spojování symbolů

Poté, co jsou symboly přetaženy na pracovní plochu, je nutné definovat jejich pořadí jejich spojením pomocí spojnic. Každý symbol má spojovací body znázorněné černou tečkou zobrazenou při označení symbolu. Jejich počet se liší dle typu symbolu. Aby byl diagram validní, musí každý spojovací bod obsahovat spojnici. Dále se liší jejich typ. Existují dva typy, výstupní bod, kde může spojnice pouze začínat, a vstupní bod, kde může spojnice pouze končit.

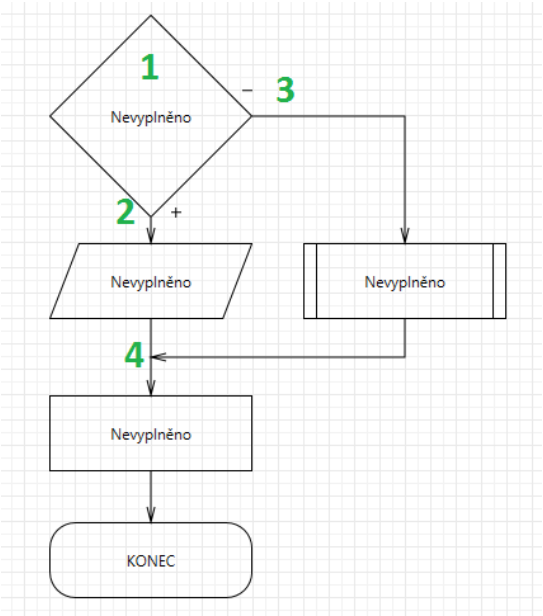


1) Symbol start obsahuje jeden výstupní bod dole.
2) Symbol pro vstup a výstup dat obsahuje jeden vstupní bod nahoře a jeden výstupní bod dole.
3) Symbol zpracování dat obsahuje jeden vstupní bod nahoře a jeden výstupní bod dole.
4) Symbol konec obsahuje jeden vstupní bod nahoře.
5) Symbol cyklu obsahuje dva vstupní body, jeden nahoře a jeden vlevo a dva výstupní body, jeden vpravo a jeden dole.
6) Symbol funkce obsahuje jeden vstupní bod nahoře a jeden výstupní bod dole.
7) Symbol podmínky obsahuje jeden vstupní bod nahoře a dva výstupní body vpravo a dole.

Spojování symbolů musí být zahájeno vždy ve výstupním bodu nějakého symbolu, zahájení je provedeno klikem levým tlačítkem myši na spojovací bod a ukončeno klikem na nějaký vstupní bod nějakého symbolu. Při aktivním spojování a kliku na prázdnou pracovní plochu se vytvoří koleno.

5.2.1 Větvení programu

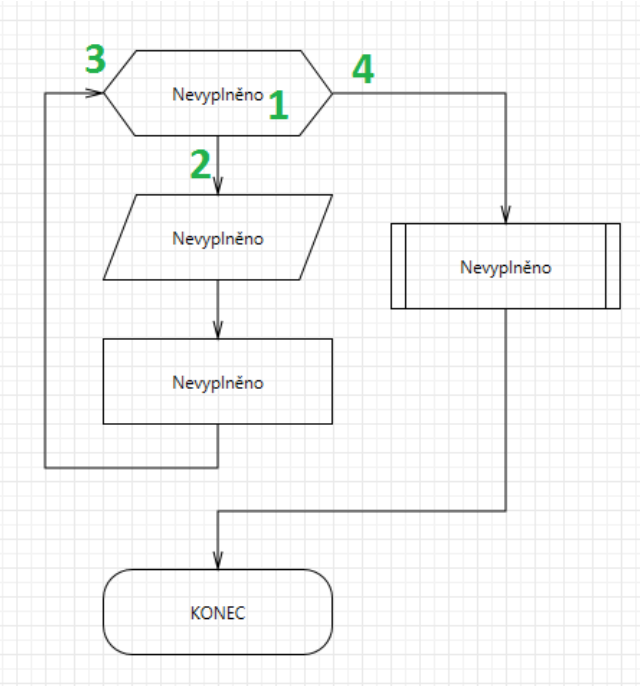
Větvení programu je definováno symbolem podmínky, který má dva výstupní body, v každém z nich začíná nová větev. Ukončení větvení musí být definováno spojením dvou větví tak, že jedna spojnice není ukončena ve vstupním bodě nějakého symbolu, ale na nějaké spojnici druhé větve.



1) Symbol podmínky uvozující větvení
2) První větev, vykonaná při splnění podmínky
3) Druhá větev, vykonaná při nesplnění podmínky
4) Ukončení větvení

5.2.2 Tvorba cyklů

Symbol cyklu slouží k definování opakujících se akcí. Opakují se akce, které jsou obsaženy v těle cyklu. Tělo cyklu je posloupnost příkazů mezi výstupním bodem dole a vstupním bodem vlevo. Algoritmus dále pokračuje z výstupního bodu vpravo.

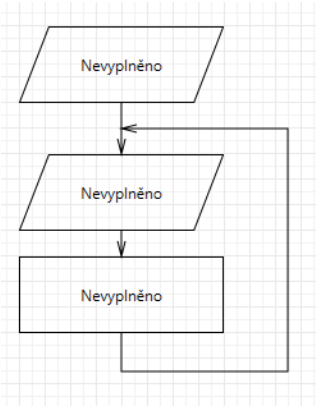


1) Symbol cyklu
2) Tělo cyklu, příkazy, které jsou napojeny na první výstupní bod, se budou opakovat
3) Ukončení těla cyklu ve vstupním bodě
4) Pokračován algoritmu v druhém výstupním bodě, další příkazy již nejsou součástí cyklu

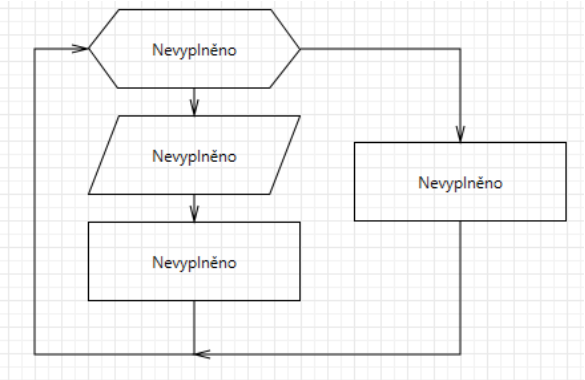
5.2.3 Omezení pří spojování

Spojování symbolů má jasně definovaná pravidla.

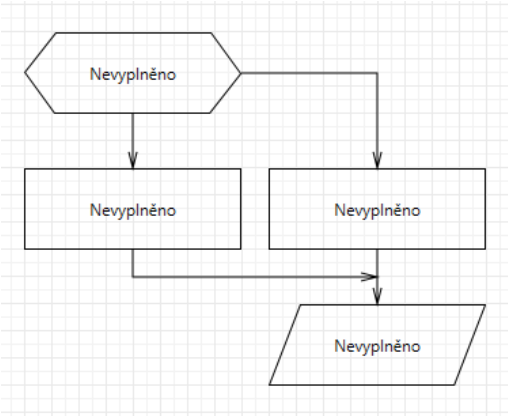
- Každý spojovací bod musí obsahovat právě jednu spojnici
- Nelze napojit spojnici na symbol, který je již součástí diagramu



- Nelze napojovat spojnice na těla cyklů



- Nelze ukončit tělo cyklu jinde, než v bodě k tomu určeném

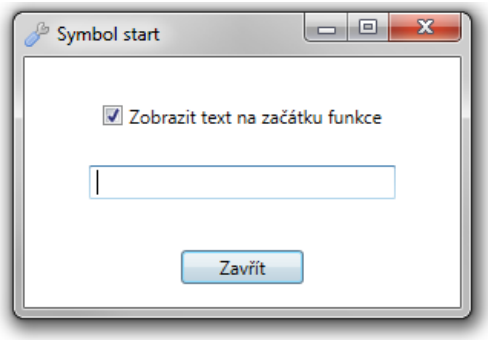


5.3 Definice vlastností symbolů

Každý symbol má jasně definované vlastnosti, které mu může uživatel přiřadit. Tyto vlastnosti definují operaci, kterou symbol představuje. Definice vlastností se provádí v okně, které je vyvoláno dvojklikem na symbol na pracovní ploše, nebo jeho označením a stiskem klávesy *enter*. Součástí většiny oken je také řádek zobrazující náhled zvolené funkce. Pokud je vlastností výběr proměnné, je někdy možné zadat i přímou hodnotu, což se řídí definovanými pravidly.

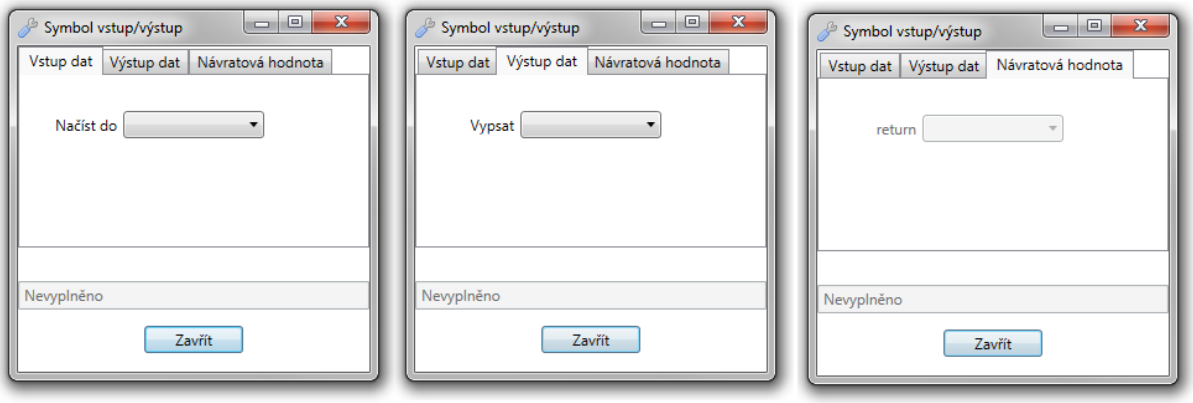
5.3.1 Symbol start

Jedinou vlastností, přiřaditelnou symbolu start, je možnost vypsání textu při začátku algoritmu funkce.



5.3.2 Symbol vstupu a výstupu dat

Symbol pro vstup a výstup dat může mít tři funkce. První je vstup dat, kdy uživatel musí vybrat proměnnou, do které budou vstupní data uložena, druhou je výstup dat, kdy uživatel musí buď vybrat proměnnou, jejíž obsah bude vypsán, nebo zadat přímou hodnotu k vypsání. Tato hodnota se řídí pravidly pro zadávání přímých hodnot. Třetí funkcí je definice návratové hodnoty funkce, kdy uživatel buď vybere proměnnou, nebo zadá přímou hodnotu.

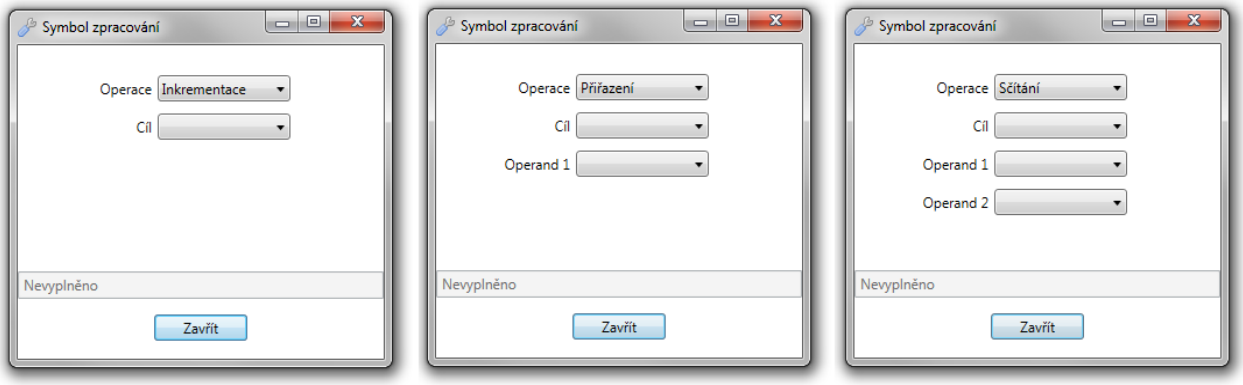


5.3.3 Symbol zpracování

Symbol zpracování definuje jednu nebo dvě proměnné popřípadě přímé hodnoty a operaci, která se s těmito prvky má provést. Dostupné jsou tyto operace, v závorce počet parametrů, 3 – cíl a dva operandy, 2 – cíl a jeden operand, 1 – operace pouze s jednou proměnnou

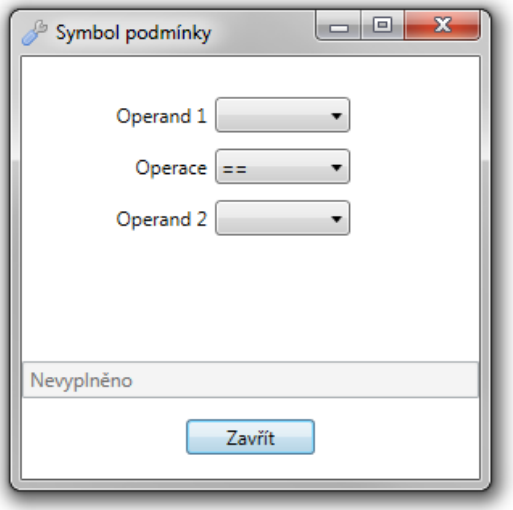
- Sčítání (3)
- Odčítání (3)
- Násobení (3)
- Dělení (3)
- Zbytek po celočíselném dělení (modulo) (3)
- Přiřazení hodnoty do proměnné (2)
- Vyhodnocení délky pole nebo řetězce (2)
- Definice délky pole (2)
- Přiřazení do proměnné s konverzí datového typu (2)
- Spojení dvou parametrů do řetězce (3)
- Získání jednoho znaku na konkrétní pozici v řetězci (3)
- Vygenerování náhodného čísla (1)
- Inkrementace (1)
- Dekrementace (1)
- Výpočet sinu (2)
- Výpočet cosinu (2)

- Výpočet tangent (2)
- Výpočet kotangent (2)
- Druhá odmocnina (2)



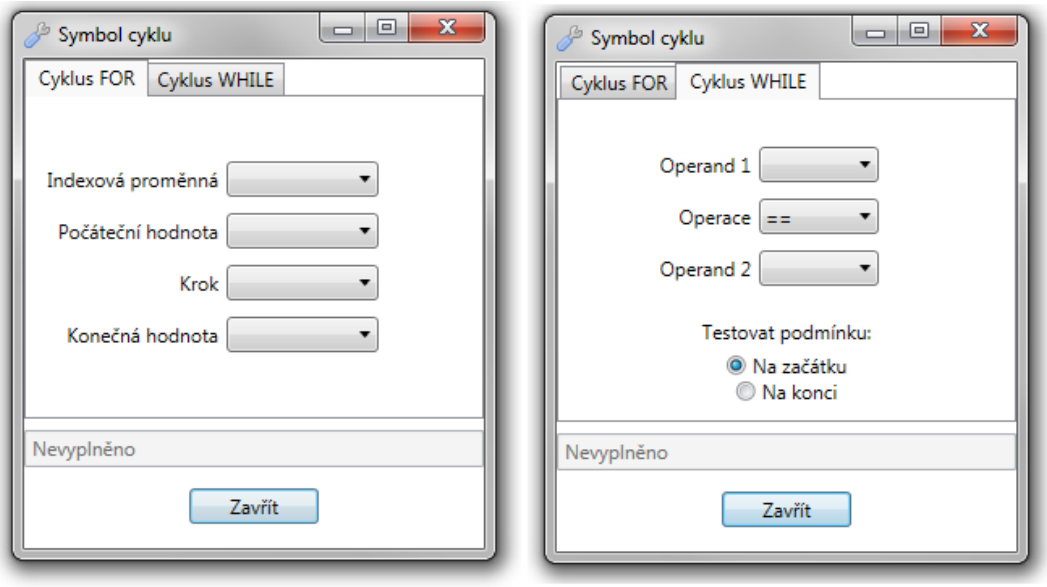
5.3.4 Symbol podmínky

Podmínka je definována dvěma operandy a operací porovnání, která se na nich provede. V případě pravdivosti operace se provede větev se symbolem +, jinak větev se symbolem -.



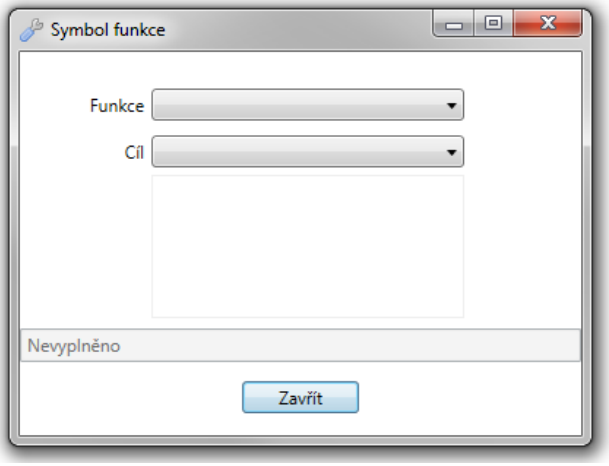
5.3.5 Symbol cyklu

Dostupné jsou dva typy cyklů. Cyklus s pevným počtem opakování, kde je použita indexová proměnná, jejíž obsah se mění dle parametrů, s každou změnou se provede jedna iterace cyklu. Parametry jsou počáteční hodnota, krok změny, konečná hodnota. Druhým typem je cyklus, který nutnost další iterace vyhodnocuje dle podmínky. Podmínka může být na začátku iterace či na konci. Parametry jsou poté stejné jako u symbolu podmínky, podmínka je definována dvěma operandy a operací porovnání, která se na nich provede. Pokud je podmínka vyhodnocena jako pravdivá, cyklus provede další iteraci, pokud ne, cyklus skončí.



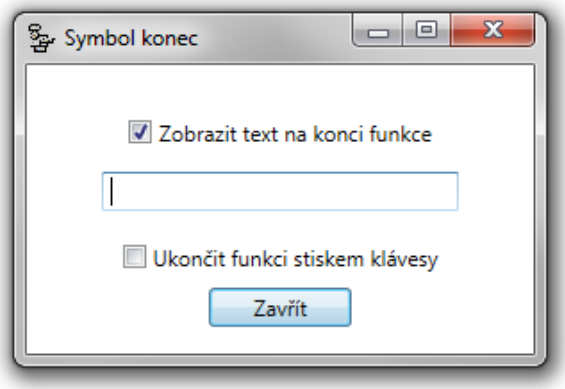
5.3.6 Symbol funkce

Symbol funkce slouží k použití vytvořených funkcí v algoritmu. Jeho operandy se liší dle použité funkce a jsou jimi proměnná, do které se uloží návratová hodnota a poté proměnné, nebo přímé hodnoty definující parametry funkce.



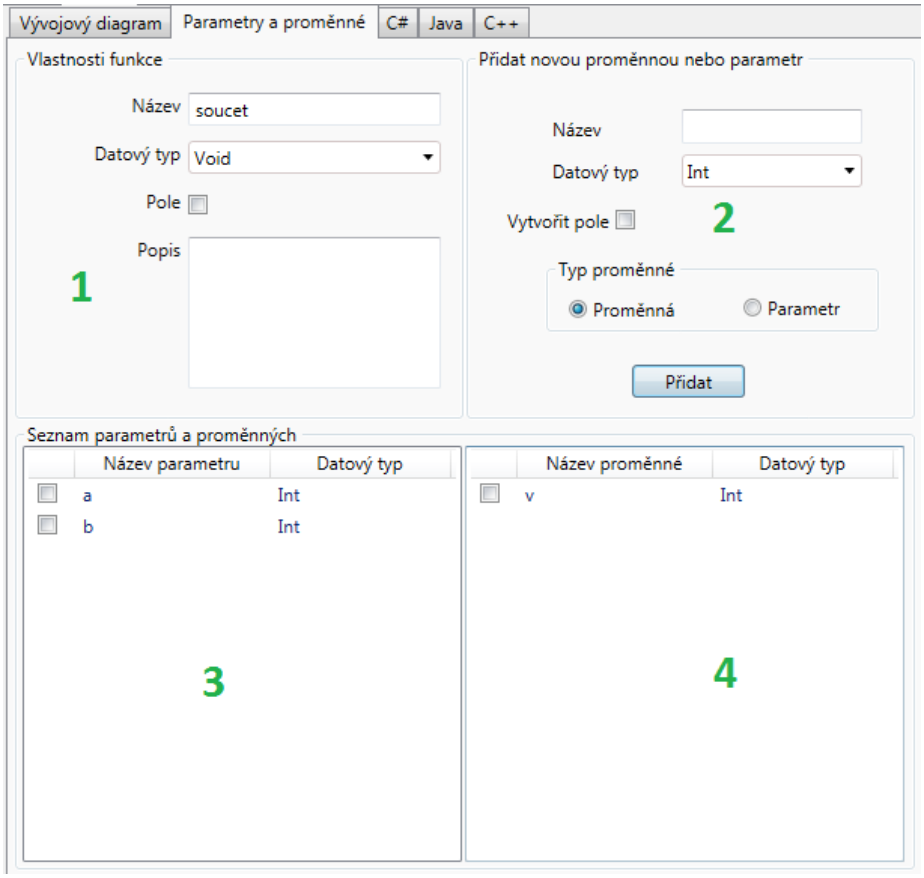
5.3.7 Symbol konec

Parametry nastavitelné tomuto symbolu jsou dva. Prvním z nich je možnost vypsání textu na konci algoritmu funkce a druhým je možnost nastavení čekání na stisk klávesy.



5.4 Definice vlastností funkcí

Každá funkce je určena definovanou množinou vlastností. Tyto vlastnosti se zadávají na druhé kartě rozhraní funkce.



Rozhraní je rozděleno na čtyři části

1) Oblast pro základní vlastnosti funkce
    - Název funkce, který se musí řídit pravidly pro zadávání názvů
    - Datový typ návratové hodnoty
    - Možnost, aby návratová hodnota byla pole daného datového typu
    - Slovní popis vytvořené funkce

2) Oblast pro tvorbu proměnných nebo parametrů (dle výběru)
3) Seznam vytvořených parametrů
4) Seznam vytvořených proměnných

6 Proměnné

Proměnná je pojmenované místo v paměti, které slouží k uchovávání dat. Proměnná má definovaný datový typ, název a implicitní hodnotu danou datovým typem. Jsou dostupné tři typy proměnných

1) Globální proměnné, definované ve vlastnostech projektu, dostupné ze všech funkcí aplikace
2) Lokální proměnné, definované ve vlastnostech funkcí, dostupné pouze pro jednu danou funkci, ve které byly definovány
3) Parametry funkcí, speciální proměnné, sloužící k předávání dat do volaných funkcí

6.1 Datové typy proměnných

Každá proměnná nebo návratový typ funkce má svůj definovaný datový typ. Je dostupných 6 datových typů:

- *Integer* pro celá čísla
- *Double* pro desetinná čísla
- *String* pro textové řetězce
- *Char* pro jednotlivé znaky
- *Boolean* pro logické hodnoty
- *Void* pouze pro návratové hodnoty, kde symbolizuje, že funkce nemá návratovou hodnotu

6.2 Pravidla pro zadávání názvů proměnných

Zadávání názvů proměnných má dvě pravidla, text se může skládat pouze ze znaků anglické abecedy, čísel a znaku podtržítko. Prvním znakem názvu musí být znak anglické abecedy, nebo podtržítko. Názvem nemůže být rezervované jméno jako například „*for*“, „*int*“, „*if*“ atd. Tato pravidla jsou ošetřena validací při zadávání a chyba je zobrazena v kontextové nápovědě.

6.3 Speciální proměnné

Speciální proměnné jsou globální proměnné, která aplikace vytvoří sama. Takovými proměnnými jsou u konzolových projektů:

- Tlac1, Tlac2 (*boolean*) pro detekci stisku tlačítek
- LED_Y, LED_R, LED_G (*boolean*) pro ovládání diod
- REPRO (*boolean*) pro ovládání reproduktoru
- BULB (*boolean*) pro ovládání žárovky

- STISKNUTO (*boolean*) pro detekci stisku klávesy na klávesnici
- CTI_KLAV (*char*) pro získání stisknutého znaku na klávesnici
- LEDBAR (*int*) pro odeslání hodnoty k zobrazení na *LedBaru*
- OUTPUT (*int*), hodnoty 0 pro výstup na display (*implicitně*), 1 pro výstup na sériovou linku

U projektů s grafickým rozhraním to jsou proměnné pro ovládání některých vlastností grafických komponent. Názvy těchto proměnných mají tvar *jméno_komponenty.vlastnost*. Tyto vlastnosti jsou:

- *IsVisible* (*boolean*) určuje viditelnost prvku v prostředí
- *Content, Text* (*string*) určuje textový obsah prvku
- *IsEnabled* (*bolean*) určuje aktivitu prvku
- *IsChecked* (*boolean*) určuje označení prvku (zaškrtnutí)
- *Value* (*double*) určuje hodnotu prvku, například naplnění ukazatele průběhu

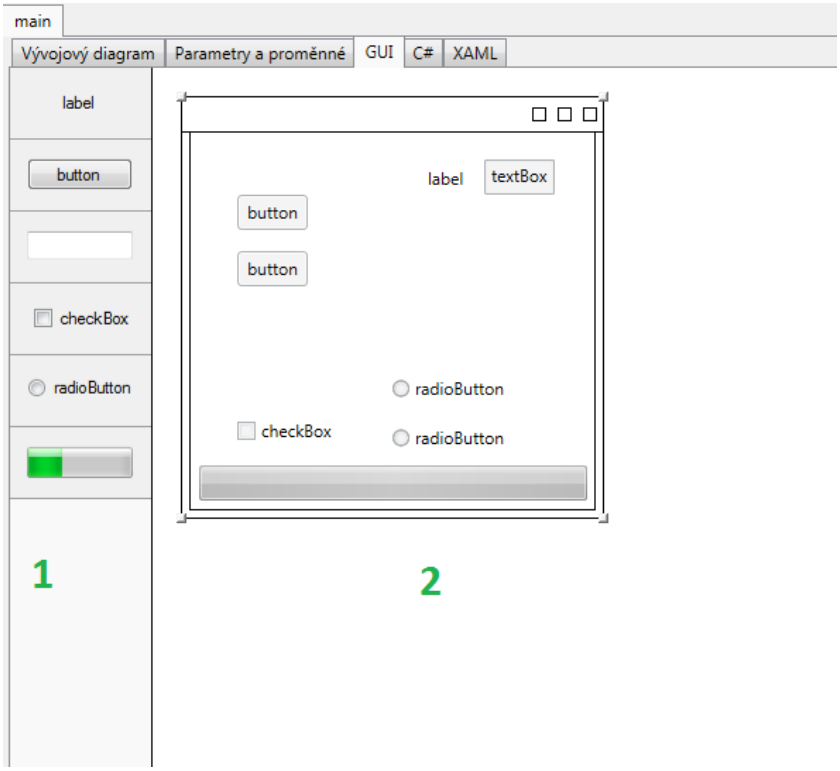
6.4 Pravidla pro zadávání přímých hodnot

Zadávání přímých hodnot, pokud je tato možnost přístupná, je aktivováno dvojklikem na zvolené pole. Tvar přímé hodnoty se řídí dle očekávaného datového typu. Zadávání je ošetřeno validací a v případě chyby je zobrazena kontextová nápověda. Tvary vstupu dle datových typů:

- *integer*
  - Pokud je očekáván datový typ *integer*, musí být zadáno pouze celé číslo. Vstup nesmí obsahovat jiné znaky než čísla a nesmí zůstat prázdný.
- *Double*
  - Pokud je očekáván datový typ *double*, musí být zadáno pouze desetinné nebo celé číslo. Vstup nesmí obsahovat jiné znaky než čísla, případně tečku a nesmí zůstat prázdný.
- *String*
  - Pokud je očekáván datový typ *string*, vstupem může být jakýkoliv řetězec, který je ohraničený znakem uvozovka. Např. „xxx“ nebo „12 aaa“…
- *Char*
  - Pokud je očekáván datový typ *char*, vstup musí být tvaru ‘_‘, kde podtržítko může být nahrazeno jakýmkoliv znakem.
- *Boolean*
  - Pokud je očekáván datový typ *boolean*, vstupem může být pouze hodnota *true* (*True*), nebo *false* (*False*).

7 Tvorba grafických rozhraní aplikací

Pokud je tvořený projekt projektem s grafickým rozhraním, jednou ze záložek vlastností funkce *Main* je záložka GUI, která obsahuje editor grafického rozhraní. Editor grafického rozhraní slouží k vytvoření grafického uživatelského rozhraní aplikace a následnému navázání vytvořených funkcí na události vyvolané tímto rozhraním.



Editor je rozdělen na dvě části

1) Lišta s dostupnými grafickými prvky
2) Pracovní plocha s tvořeným rozhraním aplikace

7.1 Dostupné grafické prvky

7.1.1 Window

Prvek *Window*, neboli okno, není dostupný z lišty komponent, ale je automaticky vytvořen při startu. Představuje okno aplikace a plochu, na kterou lze umisťovat další komponenty.

7.1.2 Label

Label je jednoduchý textový popisek, který lze umístit kamkoliv do okna

7.1.3 Button

Button je tlačítko, jehož hlavní funkce je reakce na stisk

7.1.4 TextBox

TextBox je prvek pro zadávání textového vstupu

7.1.5 CheckBox

CheckBox je zaškrtávací tlačítko

7.1.6 RadioButton

RadioButton je přepínací tlačítko

7.1.7 ProgressBar

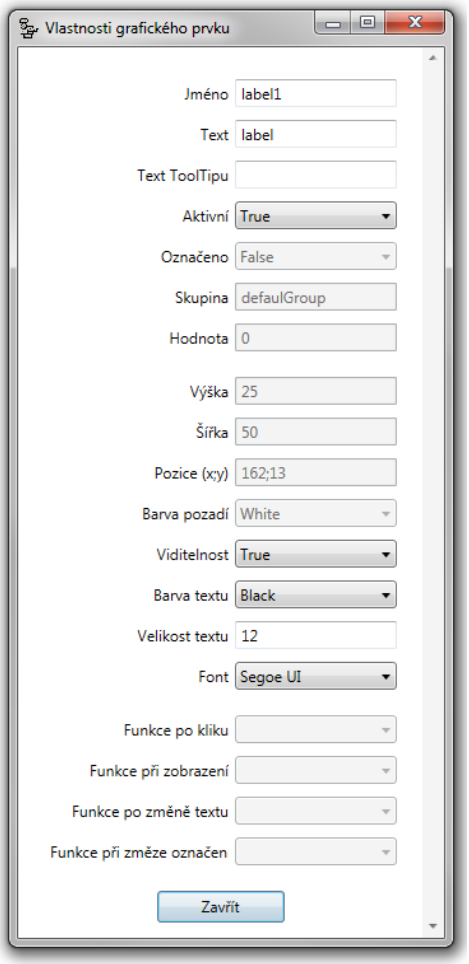
ProgressBar je indikátor průběhu

7.2 Pracovní plocha

Pracovní plocha slouží k umisťování komponent. Umisťování je možné na jakoukoliv pozici uvnitř okna se zaokrouhlováním na 5px. Jakoukoliv komponentu je možné zvětšit či zmenšit pomocí čtverečků v rozích.

7.3 Vlastnosti grafických komponent

Dvojklikem na jakoukoliv grafickou komponentu se otevře okno s jejími vlastnostmi, které má uživatel možnost změnit.



Dostupnost vlastností se řídí právě zvolenou komponentou.

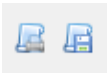
- Jméno – jasně definuje jméno komponenty, řídí se pravidly pro jména a musí být unikátní v rámci aplikace
- Text – pokud je součástí grafiky text, toto pole slouží pro jeho definici
- Text *ToolTipu* – definuje kontextovou nápovědu, vyvolanou po najetí myší na komponentu
- Aktivní – určuje, zda je komponenta dostupná pro interakci s uživatelem. Pokud je neaktivní, zešedne
- Označeno – vlastnost určující aktivitu zaškrtávacího a přepínacího tlačítka
- Skupina – vlastnost určená pouze pro přepínací tlačítka, z tlačítek ve stejné skupině může být zaškrtnuto vždy pouze jedno
- Hodnota – určuje číselnou hodnotu například ukazatele průběhu
- Šířka a výška – nejsou dostupné pro editaci, jejich změna se provádí pomocí změny rozměrů komponenty pomocí myši
- Pozice – také není dostupná k editaci, provádí se umisťováním komponent pomocí myši
- Barva pozadí
- Viditelnost – určuje, zda je komponenta viditelná pro uživatele

- Barva textu – určuje barvu textu komponent s textovým polem
- Velikost textu
- Font
- Funkce po kliku – výběr funkce, která se vyvolá při kliku myší na komponentu
- Funkce při zobrazení – výběr funkce, která se vyvolá při zobrazení komponenty
- Funkce při změně textu – výběr funkce, která se vyvolá při změně textu komponenty
- Funkce při změně označení – výběr funkce, která se vyvolá při změně označení komponent

7.4 Použití grafických komponent

Některé vlastnosti grafických komponent lze nastavovat i programově. Každá komponenta po svém vytvoření přidá několik globálních proměnných, pomocí kterých se tyto vlastnosti ovlivňují. U všech komponent je to vlastnost viditelnost, dále to je dostupnost, označení, hodnota, nebo text. Vytvořené globální proměnné mají tvar *nazev_komponenty.vlastnost*.

8 Práce se zdrojovým kódem

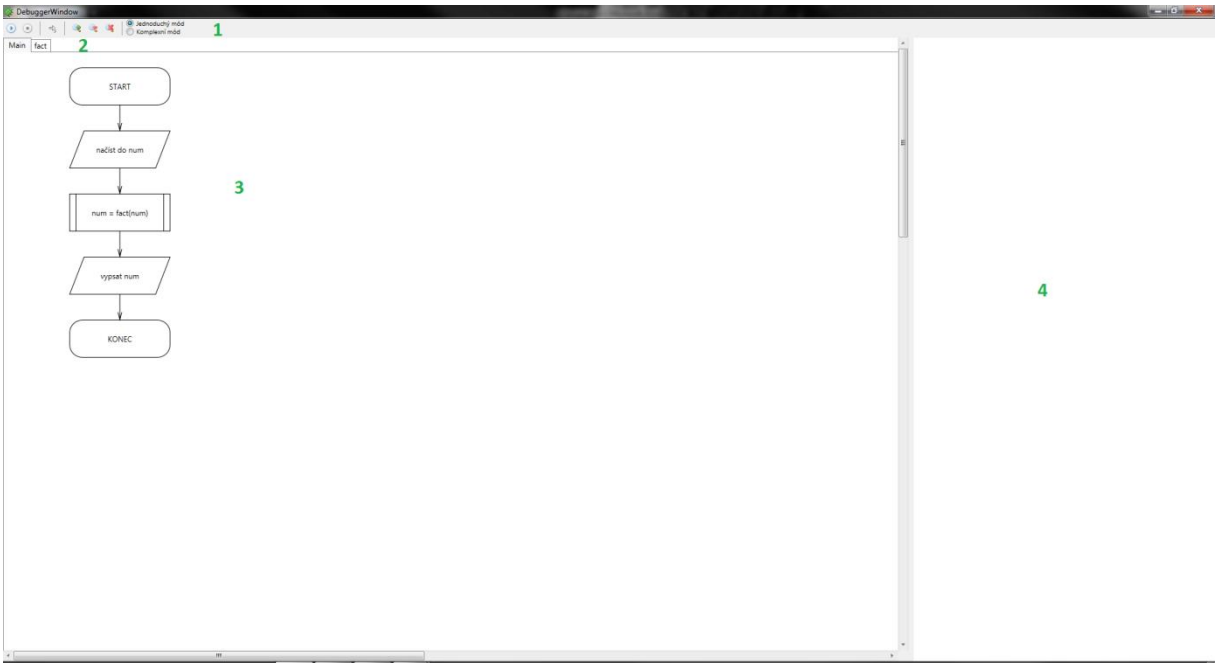
Aplikace umí z validního vytvořeného projektu vygenerovat zdrojové kódy v různých jazycích. Zdrojové kódy mají zvýrazněnou syntaxi a čísla řádků. Uživatel má možnost vygenerovaný zdrojový kód uložit jako textový soubor, vytisknout, nebo exportovat do souboru pomocí ikon z panelu nástrojů. 

8.1 Tisk

Funkce tisk je přístupná, pouze pokud je otevřen tisknutelný obsah (diagram nebo zdrojový kód) z hlavního menu položkou Soubor – tisk, z panelu nástrojů pomocí ikon s tiskárnou, nebo klávesovou zkratkou *Control+P*. Tisk zdrojových kódů neumožňuje žádná nastavení, mimo standardního tiskového dialogu Windows. Tisk vývojových diagramů probíhá ve vlastním rozhraní, které umožňuje nastavení přiblížení, papíru, orientace a poskytuje základní informace o tiskové úloze. Další nastavení jsou možná pomocí standardního tisového dialogu Windows.

9 Simulátor

Aplikace umožňuje jednoduchou simulaci vytvořených validních konzolových projektů. Simulátor je vyvolán z panelu nástrojů ikonou  , nebo klávesovou zkratkou *Alt+S*. Po vyvolání příkazu simulace dojde ke kontrole validity projektu, kompilaci a spuštění rozhraní simulátoru. Po spuštění je simulátor otevřen v tzv. *jednoduchém módu*. 



Rozhraní je rozděleno na několik částí:

1) Nástrojová lišta
2) Záložky pro všechny funkce projektu
3) Vývojové diagramy funkcí
4) Prostor pro zobrazení aktuálního stavu proměnných

9.1 Lišta nástrojů

Úplně poslední položkou nástrojové lišty je ovládání módu zobrazení simulátoru, kde si uživatel může vybrat komplexnost rozhraní.

Nástrojová lišta je rozdělena pomocí oddělovačů na několik částí.

9.1.1 Funkce pro zahájení a ukončení simulace

Tato část je stejná pro oba módy zobrazení a obsahuje dvě ikony

- zahájení simulace (klávesová zkratka F5)
- ukončení simulace

9.1.2 Funkce pro ovládání simulace

Tato část obsahuje funkce pro řízení chodu simulace, v jednoduchém módu obsahuje

- krok simulace na pozici dalšího symbolu

Při přepnutí do komplexního módu se možnosti rozšíří o

- krok na další řádek zdrojového kódu (klávesová zkratka F9)
- krok do aktuální funkce (klávesová zkratka F10)
- krok přes aktuální funkce (klávesová zkratka F11)
- krok ven z aktuální funkce (klávesová zkratka F12)

9.1.3 Funkce pro automatickou simulaci

Tato část je dostupná pouze v komplexním módu a umožňuje automatické krokování algoritmu v časovém intervalu.

- zahájení automatického krokování
- ukončení automatického krokování
- 

nastavení časového intervalu

9.1.4 Funkce pro přibližování a oddalování

Ikony s lupou umožňují přibližovat či oddalovat zobrazené vývojové diagramy, či zvětšovat nebo zmenšovat velikost textu zdrojového kódu.

9.2 Komplexní mód

Při přepnutí do komplexního módu se nejen rozšíří možnosti simulace, ale je zobrazen i zdrojový kód simulované aplikace s vyznačenou aktuální pozicí.